1

A Practical Construction for Decomposing Numerical Abstract Domains

ANONYMOUS AUTHOR(S)

Numerical abstract domains such as Polyhedra, Octahedron, Octagon, Interval, and others are an essential component of static program analysis. The choice of domain offers a performance/precision tradeoff ranging from cheap and imprecise (Interval) to expensive and precise (Polyhedra). Recently, significant speedups were achieved for Octagon and Polyhedra by manually decomposing their transformers to work with the Cartesian product of projections associated with partitions of the variable set. While practically useful, this manual process is extremely time consuming, error-prone and has to be applied from scratch for every domain.

In this paper, we present a novel approach that can soundly decompose any sub-polyhedra domain. Unlike prior work, the method is generic in nature and does not require changes to the original abstract transformers or additional manual effort per domain. Further, it presents guarantees on the partitions achievable by each decomposed transformer. In general, our method achieves finer partitions than prior work.

We implemented our approach and applied it to the domains of Zones, Octagon, and Polyhedra. We then compared the performance of the decomposed transformers obtained with our generic method versus state-of-the art PPL and the faster ELINA (which uses manual decomposition). Against the latter we demonstrate finer partitions and an associated speedup of about 2x on average. Our results indicate that the construction presented in this work is a viable method for improving the performance of numerical domains. It enables designers of abstract domains to benefit from decomposition without re-writing all of their transformers from scratch (as required by prior methods).

CCS Concepts: •**Theory of computation → Program verification; Program analysis; Abstraction;**

Additional Key Words and Phrases: Abstract Interpretation, Numerical Domains, Domain Decomposition

ACM Reference format:
Anonymous Author(s). 2017. A Practical Construction for Decomposing Numerical Abstract Domains. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 27 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Numerical abstract domains are a key component of modern static program analyzers (Blanchet et al. 2003; Gurfinkel et al. 2015). The design of these domains remains an art as designers of domains are faced with two critical choices while fine-tuning the cost and the precision of their domain. These are: (a) the shape of constraints which determines the domain's expressivity, and (b) the precision and scalability of the abstract transformers.

Improving scalability of abstract transformers is an inherently hard problem as limiting the shape of the constraints allowed in the domain does not necessarily guarantee reduction in the transformer's asymptotic complexity. Indeed, the most precise transformer for assignments in weakly relational domains such as Octagon (Miné 2006), Zones (Miné 2002) and TVPI (Simon and King 2010) have the same worst-case exponential complexity as the transformers in the most expensive Polyhedra (Cousot and Halbwachs 1978) domain.

To improve scalability of the overall analysis, designers of abstract domains may introduce approximations (of the best transformer) with the hope of improving performance in practical

A note.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

scenarios while maintaining sufficient precision needed to verify the property of interest. Because of the importance of scaling the analysis to realistic applications, there has been increased interest in improving the performance of existing domains. Existing approaches can be roughly divided into two classes: (a) implement less precise transformers tuned to the specific verification task (Blanchet et al. 2003; Heo et al. 2016; Venet and Brat 2004), or (b) maintain the same precision as the existing implementation and improve performance by designing specialized algorithms and data structures optimized for the particular domain (Gange et al. 2016; Jourdan 2017). While challenging to devise, the latter approach is appealing because it does not explicitly lose precision like (a) yet increases performance.

A technique for achieving this goal is the concept of decomposition. It is based on the observation that abstract elements may be decomposed into Cartesian products over independent subsets of variables; hence, a given domain transformer does not need to be applied on the complete abstract element but rather on some part of it, thus reducing cost. The first attempt at decomposition was for the Polyhedra (Halbwachs et al. 2006, 2003) domain where the abstract elements were decomposed based on partitioning a variable set into subsets such that constraints exist only between the variables in the same subset. The partitioning was performed on the fly, however the partitions produced were too coarse.

Recently, the concept of online decomposition where the partitions are maintained and updated based on the transformer semantics has been applied to achieve speed-ups by orders of magnitude over standard implementations for the Octagon (Singh et al. 2015) and the Polyhedra (Singh et al. 2017) domain. However, in both cases the decomposition was manually designed from scratch for the standard transformers of the particular domain. The downside of this approach is that the substantial effort invested in decomposing the transformers of the particular domain cannot be reused and needs to be repeated for every new domain. This task is extremely difficult and error-prone as it requires devising new algorithms and data structures from scratch.

To illustrate the issue, consider an element $\mathcal{I} = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0\}$ in the Octagon domain which captures constraints of the form $\pm x_i \pm x_j \leq c$ between the program variables and the conditional statement $x_2 + x_3 + x_4 \leq 1$. There are multiple ways to implement a sound conditional transformer in the Octagon domain for the given conditional statement. One may define a sound conditional transformer T_1 that adds the non-redundant constraint $-x_1 - x_4 \leq 1$ to $\mathcal{I}$ resulting in the output $\mathcal{I}' = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0, -x_1 - x_4 \leq 1\}$ whereas another transformer T_2 may add $-x_2 - x_3 \leq 1$ to $\mathcal{I}$ resulting in $\mathcal{I}'' = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0, -x_2 - x_3 \leq 1\}$. The set of variables in the constraints added by the two transformers are disjoint. The specialized decomposition for the Octagon domain (Singh et al. 2015) requires access to the exact definition of the transformer, i.e., it will produce different decomposition for T_1 and T_2.

This Work. The key objective of this work is to bring the power of decomposition to all numerical (sub-polyhedra) domains without requiring complex manual effort from the domain designer. This would enable domain designers to achieve speed-ups without requiring them to re-write all of their abstract transformers from scratch each time.

More formally, our goal is to provide a systematic correct-by-construction method that given an abstract transformer T in a sub-polyhedra domain (e.g., Zones), generates a decomposed version of T that is faster than T and *does not* require any change to the internals of T. In this paper we present a construction that achieves this objective and show that it leads to decomposed transformers that are faster than prior, hand-tuned decomposed implementation of domains (Singh et al. 2015, 2017).

Main Contributions. Our paper makes the following contributions:

- For designers of new transformers in existing or future numerical domains, we provide specifications on how to achieve and maintain decomposition. The specifications are based on static criteria that a transformer should satisfy (Section 5).
- We introduce a general construction for obtaining decomposed transformers of existing numerical domains. This includes guarantees on the achievable decomposition (i.e., which granularity is possible) (Section 6).
- We applied our method to decompose standard end-to-end implementations of three popular and expensive domains: Polyhedra, Octagons, and Zones. The existing implementation of non-decomposed transformers in these domains did not require any modification. In some cases, our decomposed transformers are more precise than the original non-decomposed transformers.
- We evaluated the effectiveness of our decomposed analysis against state-of-the-art implementations on large real-world benchmarks including Linux device drivers. Our evaluation shows up to 6x and 2x speedups on the overall end-to-end Polyhedra and Octagon domain analysis over state-of-the-art, manual decomposition tuned to the particular implementations of these domains. For Zones, we achieve speedups of about 2 to 5x compared to our own, non-decomposed implementation. All speedups are due to our method and not due to implementation techniques.

2 GENERIC MODEL FOR NUMERICAL ABSTRACT DOMAINS

An abstract domain consists of a set of abstract elements and a set of transformers that model the effect of program statements and expressions (assignment, conditionals, etc.) on the abstract elements. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of variables. In this paper, we consider sub-polyhedra domains, i.e., numerical abstract domains $\mathcal{D}$ that encode linear relationships between the variables in $\mathcal{X}$ of the form:

$$\sum_{i=1}^{n} a_i x_i \otimes c, \qquad \text{where } x_i \in \mathcal{X}, a_i \in \mathbb{Z}, \otimes \in \{\leq, =\}, c \in \mathcal{C}. \tag{1}$$

Typical choices for $\mathcal{C}$ include $\mathbb{Q}$ (rationals) and $\mathbb{R}$ (reals). As with any abstraction, the design of a numerical domain is guided by the cost vs. precision tradeoff. For instance, the Polyhedra domain (Cousot and Halbwachs 1978) is the most precise numerical domain yet is also the most expensive. On the other hand, the Interval (Box) domain is cheap but is also very imprecise as it does not preserve relational information between variables. Between these two sit a number of domains with varying degrees of precision and cost; examples include Two Variables Per Inequality (TVPI) (Simon and King 2010), Octagons (Miné 2006), and Zones (Miné 2002).

Representing domain constraints. We introduce notation for describing the set of constraints a given domain $\mathcal{D}$ can express for variables $\mathcal{X}$. This set of constraints is referred to as $\mathcal{L}_{\mathcal{X},\mathcal{D}}$ and is determined by four components $(n, \mathcal{R}, \mathcal{T}, \mathcal{C})$:

- The size n of the variable set $\mathcal{X}$.
- A relation $\mathcal{R} \subseteq \mathcal{R}_1 \times \mathcal{R}_2 \times \ldots \times \mathcal{R}_n$ to describe the universe of possible coefficients. Each $\mathcal{R}_i \subseteq \mathbb{Z}$ is a set of integers defining the allowed values for the coefficients a_i. Typical examples for $\mathcal{R}_i$ include $\mathbb{Z}$, $\mathbb{U} = \{-1, 0, 1\}$, and $\mathbb{L} = \{-2^k, 0, 2^k \mid k \in \mathbb{Z}\}$.
- The set $\mathcal{T} \subseteq \{\leq, =\}$ determining equality/inequality constraints.
- The set $\mathcal{C}$ containing the allowed values for the constant c in (1). Typical examples include $\mathbb{Q}$ and $\mathbb{R}$.

Table 1 shows prototype constraints allowed by different numerical domains in the above notation. The set of constraints $\mathcal{L}_{\mathcal{X},\mathcal{D}}$ representable by a domain $\mathcal{D}$ contains all constraints of the form $\sum_{i=1}^{n} a_i x_i \otimes c$ where: (i) the coefficient list of each expression $\sum_{i=1}^{n} a_i x_i$ is a permutation of a tuple in $\mathcal{R}$, (ii) $\otimes \in \mathcal{T}$, and (ii) the constant $c \in C$. For instance, the possible constraints $\mathcal{L}_{\mathcal{X},\text{octagon}}$ for the Octagon domain over real numbers are described via the tuple $(n, \mathbb{U}^2 \times \{0\}^{n-2}, \{\leq, =\}, \mathbb{R})$.

Table 1. Instantiation of constraints expressible in various numerical domains.

Domain	$\mathcal{R}$	$\mathcal{T}$	C	Reference
Polyhedra	$\mathbb{Z}^n$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Cousot and Halbwachs 1978)
Linear equality	$\mathbb{Z}^n$	$\{=\}$	$\mathbb{Q}, \mathbb{R}$	(Karr 1976)
Octahedron	$\mathbb{U}^n$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Claris and Cortadella 2007)
Stripes	$\{(a, a, -1, 0, \ldots, 0) \mid a \in \mathbb{Z}\}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Ferrara et al. 2008)
TVPI	$\mathbb{Z}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Simon and King 2010)
Octagon	$\mathbb{U}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Miné 2006)
Logahedra	$\mathbb{L}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Howe and King 2009)
Zones	$\{1, 0\} \times \{0, -1\} \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Miné 2002)
Strict upper bound	$\{1\} \times \{-1\} \times \{0\}^{n-2}$	$\{\leq\}$	$\{-1\}$	(Logozzo and Fähndrich 2008)
Interval	$\{1, -1\} \times \{0\}^{n-1}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	(Cousot and Cousot 1976)

Example 2.1. Consider a program with four variables and a fictive domain that can relate at most two:

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\} \text{ and } \mathcal{L}_{\mathcal{X},\mathcal{D}} : (4, \mathbb{U}^2 \times \{0\}^2, \{\leq, =\}, \{1, 2\}).$$

Here, the constraint $2x_1 + 3x_4 \leq 2 \notin \mathcal{L}_{\mathcal{X},\mathcal{D}}$ as no permutation of tuples in $\mathbb{U}^2 \times \{0\}^2$ can produce $(2, 0, 0, 3)$. Similarly, $x_2 - x_3 \leq 3 \notin \mathcal{L}_{\mathcal{X},\mathcal{D}}$ even though there exists a permutation of tuples in $\mathbb{U}^2 \times \{0\}^2$ that can produce $(0, 1, -1, 0)$, but $3 \notin C$. However, the constraints $x_2 - x_3 \leq 1$ and $x_2 - x_3 = 2$ are in $\mathcal{L}_{\mathcal{X},\mathcal{D}}$.

Defining an abstract domain. An abstract element $\mathcal{I}$ in a domain $\mathcal{D}$ is a conjunction of a finite number of constraints from $\mathcal{L}_{\mathcal{X},\mathcal{D}}$. By abuse of notation we will represent $\mathcal{I}$ as a set of constraints (interpreted as a conjunction of the constraints in the set). The set of all possible abstract elements is denoted by $\mathcal{P}_{\mathcal{D}}$ and typically forms a lattice $(\mathcal{P}_{\mathcal{D}}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ with respect to the domain order $\sqsubseteq$. Given abstract elements $\mathcal{I}$ and $\mathcal{I}'$, $\mathcal{I} \sqcup \mathcal{I}'$ is the smallest element approximating the union $\mathcal{I} \cup \mathcal{I}'$ of the polyhedra and is computed by the join transformer. Similarly $\mathcal{I} \sqcap \mathcal{I}' = \mathcal{I} \cap \mathcal{I}'$ is the meet transformer. There are usually 40 abstract transformers in a given domain $\mathcal{D}$. While our theory handles all 40 transformers, we focus on the join ($\sqcup$), meet ($\sqcap$), conditional, assignment, and widening ($\triangledown$) transformers in this paper. We chose these because they are the most expensive transformers in the domain and thus their design shows the most variation, i.e., they can be implemented in multiple ways. We note that there is an equivalent representation of an abstract element based on the generator representation where the element is encoded as a collection of vertices, rays and lines. In this paper, we use the constraint representation as it leads to a clearer exposition of the ideas. However, our technical results are also valid with the generator representation.

As standard, we use the concretization function γ to denote with $\gamma(\mathcal{I})$ the concrete element (polyhedron) represented by the abstract element $\mathcal{I}$. We note that in the constraint representation, it is possible for $\mathcal{I}$ to include redundant constraints, that is, removing a constraint from $\mathcal{I}$ may not change the represented concrete element $\gamma(\mathcal{I})$. Further, the minimal representation of a concrete element $\gamma(\mathcal{I})$ is not unique as there could be two non-comparable abstract elements $\mathcal{I}$ and $\mathcal{I}'$ where $\gamma(\mathcal{I}) = \gamma(\mathcal{I}')$:

Example 2.2. $\mathcal{I} = \{x_1 = 0, x_2 = 0\}$ and $\mathcal{I}' = \{x_1 = 0, x_2 = 0, x_1 = x_2\}$ represent the same concrete element $\gamma(\mathcal{I})$ in the Polyhedra domain. However, $\mathcal{I}'$ contains the redundant constraint $x_1 = x_2$. $\mathcal{I}$ is not the only minimal representation as $\mathcal{I}'' = \{x_1 = 0, x_1 = x_2\}$ is also minimal for $\gamma(\mathcal{I})$.

We say an abstract domain $\mathcal{D}$ is *closed* for an abstract transformer[1] T iff for any abstract element $\mathcal{I}$ in $\mathcal{D}$, $\gamma(T(\mathcal{I})) = T^{\#}(\gamma(\mathcal{I}))$, where $T^{\#}$ is the corresponding concrete transformer (that is, the abstract transformer does not lose precision). The Polyhedra domain is closed for the conditional, assignment and meet transformers but it is not closed for the join transformer. All other domains in Table 1 are only closed under the meet transformer. Indeed, a crucial aspect of abstract interpretation is to permit sound approximations for transformers that are not closed.

Example 2.3. Consider the abstract element $\mathcal{I} = \{x_1 \leq 1, x_2 \leq 0\}$ in the Octagon domain. The conditional transformer T for the linear constraint $x_1 - 2x_2 \leq 0$ is not closed as the concrete element produced by $T^{\#}$ is $\mathcal{I}' = T^{\#}(\gamma(\mathcal{I})) = \{x_1 \leq 1, x_2 \leq 0, 2x_1 - x_2 \leq 0\}$. There does not exist a representation for $\mathcal{I}'$ in the Octagon domain as the constraint $2x_1 - x_2 \leq 0$ is not representable.

A useful concept in analysis (and one we refer to throughout the paper) is that of the best abstract transformer.

Definition 2.1. A (unary) abstract transformer T in $\mathcal{D}$ is best iff for any unary abstract transformer T' (corresponding to the same concrete transformer $T^{\#}$) it holds that for any element $\mathcal{I}$ in $\mathcal{D}$, T always produces a more precise result (in the concrete), that is, $\gamma(T(\mathcal{I})) \subseteq \gamma(T'(\mathcal{I}))$. The definition is naturally extended to multiple arguments.

In example 2.3, a possible sound approximation for the output in the Octagon domain is $\mathcal{I}'' = \mathcal{I}$ while the best transformer would produce $\{x_1 \leq 0, x_2 \leq 0, x_1 - x_2 \leq 0\}$.

3 DECOMPOSING ABSTRACT ELEMENTS

In this section we introduce the needed notation and concepts for decomposing abstract elements and transformers. As in (Halbwachs et al. 2003; Singh et al. 2015, 2017), our approach to decomposition is based on the observations that: (a) not all variables get related by a constraint in a given abstract element $\mathcal{I}$, and (b) the number of variables affected by a given program statement is small compared to the size n of the set of program variables $\mathcal{X}$. These observations enable us to decompose $\mathcal{I}$ into smaller pieces which, in turn, enables the decomposition of the domain transformers to reduce their complexity. The decomposition is not fixed and varies over iterations for the same element and thus needs to be determined and maintained dynamically. This results in better performance with respect to the original non-decomposed transformer.

We address the decomposition of abstract elements and transformers for $\mathcal{D}$ based on partitioning the variable set $\mathcal{X}$. The set $\mathcal{P}_{\mathcal{X}}$ consisting of all partitions of $\mathcal{X}$ forms a *partition lattice* $(\mathcal{P}_{\mathcal{X}}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The elements π of the lattice are ordered as follows: $\pi \sqsubseteq \pi'$, if every block of π is included in some block of π' (π "is finer" than π'). The lattice contains the usual *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) operators. In the partition lattice, $\top = \{\mathcal{X}\}$ and $\bot = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$.

Given an abstract element $\mathcal{I}$, we partition the set of program variables $\mathcal{X}$ into subsets $\mathcal{X}_k$ that we call *blocks* such that constraints only exist between variables in the same block. Each unconstrained variable x_i yields the singleton block $\{x_i\}$. We use $\pi_{\mathcal{I},\mathcal{D}} = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$ to denote the unique

[1]Throughout the paper we will simply use the term transformer to mean an abstract transformer.

finest such partition for an element $\mathcal{I}$. For simplicity, we usually omit $\mathcal{D}$ from the subscript and just write $\pi_{\mathcal{I}}$.

The partition $\pi_{\mathcal{I}}$ decomposes $\mathcal{I}$ into a set of smaller abstract elements $\mathcal{I}_k$ on the variables in a block $\mathcal{X}_k$ which we call *factors*. Each factor $\mathcal{I}_k \subseteq \mathcal{I}$ is defined by the constraints that exist between the variables in the corresponding block $\mathcal{X}_k$. $\mathcal{I}$ can be recovered from the set of factors by taking the union of the constraint sets $\mathcal{I}_k$.

Example 3.1. Consider the element $\mathcal{I} = \{x_1 - x_2 \leq 1, x_3 \leq 0, x_4 \leq 0\}$ in the TVPI domain

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\} \text{ and } \mathcal{L}_{\mathcal{X},\text{TVPI}} : (4, \mathbb{Z}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}).$$

Here $\mathcal{X}$ can be partitioned into three blocks with respect to $\mathcal{I}$ resulting in three factors,

$$\pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\}, \mathcal{I}_1 = \{x_1 - x_2 \leq 1\}, \mathcal{I}_2 = \{x_3 \leq 0\} \text{ and } \mathcal{I}_3 = \{x_4 \leq 0\}.$$

For a given $\mathcal{D}$, $\pi_{\perp} = \pi_{\top} = \perp = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$. More generally, if $\mathcal{I} \sqsubseteq \mathcal{I}'$, then $\pi_{\mathcal{I}'}$ may be finer as, coarser as, or not comparable with $\pi_{\mathcal{I}}$.

Different partitions for equivalent elements. To gain a deeper understanding of the issues with partitions, there are two interesting points worth noting. First, it is possible that two semantically equivalent abstract elements $\mathcal{I}, \mathcal{I}'$ in the domain have different partitions. That is, even if $\gamma(I) = \gamma(I')$, it may be the case that $\pi_{\mathcal{I}} \neq \pi_{\mathcal{I}'}$ or $\pi_{\mathcal{I}} \sqsubset \pi_{\mathcal{I}'}$:

Example 3.2. Consider $\mathcal{I} = \{x_1 \leq x_2, x_2 = 0, x_3 = 0\}$ with the finest partition $\pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3\}\}$, $\mathcal{I}' = \{x_1 \leq 0, x_2 = 0, x_3 = 0\}$ with $\pi_{\mathcal{I}'} = \{\{x_1\}, \{x_2\}, \{x_3\}\}$ and $\mathcal{I}'' = \{x_1 \leq x_3, x_2 = 0, x_3 = 0\}$ with $\pi_{\mathcal{I}''} = \{\{x_1, x_3\}, \{x_2\}\}$ in the Polyhedra domain. Here $\gamma(\mathcal{I}) = \gamma(\mathcal{I}') = \gamma(\mathcal{I}'')$, but the partitions are pairwise different.

Second, it is possible that for a given abstract element $\mathcal{I}$, there exists an equivalent element $\mathcal{I}'$ with finer partition but $\mathcal{I}'$ is not representable in the domain:

Example 3.3. Consider the fictive domain,

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\mathcal{D}} : \{4, \mathbb{U}^4, \{\leq, =\}, \{1, 3\}\},$$
$$\mathcal{I} = \{x_1 = 1, x_1 + x_2 - x_3 = 3, x_2 + x_3 + x_4 = 1\} \text{ with } \pi_{\mathcal{I}} = \{x_1, x_2, x_3, x_4\}.$$

This domain cannot represent the equivalent element $\{x_1 = 1, x_2 - x_3 = 2, x_2 + x_3 + x_4 = 1\}$ which has the partition $\{\{x_1\}, \{x_2, x_3, x_4\}\}$ that is finer than $\pi_{\mathcal{I}}$. This is because the constraint $x_2 - x_3 = 2$ is not representable in $\mathcal{D}$.

It is important we guarantee that regardless of how approximate a given transformer T is, the partition we end up computing for T is always sound (permissible) w.r.t. the output abstract element $\mathcal{I}$ produced by T. Next, we define this notion formally following (Singh et al. 2017).

Definition 3.1. A partition $\overline{\pi}$ is permissible w.r.t. an abstract element $\mathcal{I}$ if it is *coarser* than $\pi_{\mathcal{I}}$, that is, $\overline{\pi} \sqsupseteq \pi_{\mathcal{I}}$.

The variables related in $\pi_{\mathcal{I}}$ are also related in any permissible partition of $\mathcal{I}$, but not vice-versa. In example 3.1, $\{\{x_1, x_2\}, \{x_3, x_4\}\}$ is permissible w.r.t. $\mathcal{I}$ while $\{\{x_1\}, \{x_2, x_3, x_4\}\}$ is not. We will generally use $\overline{\pi}_{\mathcal{I}}$ to denote a permissible partition for $\mathcal{I}$.

4 RECIPE FOR DECOMPOSING TRANSFORMERS

One primary objective of this work is to define a mechanical recipe which takes as input a sound abstract transformer and produces as output a decomposed variant of that transformer, thus resulting in better analysis performance. In this section we describe the general recipe and illustrate its actual use.

At first glance the above challenge appears fundamentally difficult, because there are multiple ways to define a sound transformer in a domain $\mathcal{D}$. Standard implementations of popular numerical domains, e.g., Octagon, Zones, TVPI, do not necessarily implement the best transformer as it can be expensive; instead they usually approximate it. Interestingly, as pointed out earlier, such an approximation can make the associated partition coarser or finer. That is, the partitioning function is not monotone. Here is an example illustrating this point:

Example 4.1. Consider the elements $\mathcal{I} = \{x_1 \leq 0, x_2 \leq 0, x_1 - x_2 \leq 0\}$ with $\pi_{\mathcal{I}} = \{\{x_1, x_2\}\}$ and $\mathcal{I}' = \{x_1 \leq 0, x_2 \leq 0\}$ with $\pi_{\mathcal{I}'} = \{\{x_1\}, \{x_2\}\}$ in the Polyhedra domain. Here, $\gamma(\mathcal{I}) \subset \gamma(\mathcal{I}')$ and $\pi_{\mathcal{I}} \sqsupseteq \pi_{\mathcal{I}'}$. On the other hand, for the elements $\mathcal{I} = \{x_1 \leq 0, x_2 \leq 0\}$ with $\pi_{\mathcal{I}} = \{\{x_1\}, \{x_2\}\}$ and $\mathcal{I}' = \{x_1 + x_2 \leq 0\}$ with $\pi_{\mathcal{I}'} = \{\{x_1, x_2\}\}$. Now, $\gamma(\mathcal{I}) \subset \gamma(\mathcal{I}')$ but $\pi_{\mathcal{I}} \sqsubseteq \pi_{\mathcal{I}'}$.

Definition 4.1. A transformer T in $\mathcal{D}$ is decomposable w.r.t to its input $\mathcal{I}$ in $\mathcal{D}$ iff the output $\mathcal{I}'$ after applying T on $\mathcal{I}$ results in a partition where $\pi_{\mathcal{I}'} \neq \top$.

There are multiple ways to define a sound approximation of the best transformer in $\mathcal{D}$. It is possible to have two transformers T_1, T_2 in $\mathcal{D}$ on the same input $\mathcal{I}$ such that one produces $\top$ partition for the output while the other not. There are two principal ways to obtain a decomposable transformer. The first approach is to design each transformer from scratch, maintaining the (changing) partitions during analysis. The other approach is to provide a construction for decomposing existing transformers without knowing their internals. In Sections 4 and 6 we elaborate on this approach and show which partitions are achievable. We now elaborate on the steps that one needs to perform dynamically when decomposing a given transformer.

A construction for online transformer decomposition. There are four main steps for decomposing a given (decomposable) transformer:

(1) compute (if needed) partitions for the input(s),
(2) compute a partition for the output based on the statement/expression and input partition(s) of step 1,
(3) re-factor the inputs according to the computed output partition in step 2, and
(4) apply the transformer on one or more factors of the inputs from step 3.

We next describe these steps in greater detail.

In an ideal setting, one would always work with the finest partition for the inputs and the output so to (optimally) reduce the cost of the transformer. The finest partition for the inputs of a given transformer can always be computed from scratch by taking the abstract element and connecting the variables that occur in the same constraint in that element. The downside is that this computation may incur significant overhead. For example, computing the finest partition for an element in the Octagon domain from scratch has the same quadratic complexity as the conditional, meet and assignment transformers which basically nullifies potential performance gains from decomposing these transformers.

To compute the output partition, a naive way is to first run the transformer, obtain an abstract element as a result, and then compute the partition for that element. Of course, this approach is useless since running the standard transformer prevents performance gains. Thus, the challenge is to devise an approach that keeps track of the partitions dynamically without recomputing them from scratch. Indeed, in our construction we always compute a permissible partition for the output based on permissible partitions of the input, the program statement, and possibly additional information that is cheaply available. Once the output partition is obtained, the associated abstract element is computed directly in decomposed form by applying the original transformer to the factors of the input.

The third step in our construction involves refactoring the input(s) according to the output partition. For all transformers discussed in this paper, the permissible partition for the output is coarser than the permissible partition(s) for the input(s). Refactoring the inputs just means viewing them as partitioned with the (coarser, and thus permissible) output partition. This is done by collecting the constraints of blocks that get merged to one block.

The last step of the construction involves computing the output abstract element by applying the user-provided transformer on one or more factor(s) of the refactored input(s). Applying this transformer on smaller factors reduces its complexity and results in increased performance. In certain cases, the permissible partition for the output can be further refined after applying the transformer and without adding significant overhead. We identify such cases in Section 6.

Our construction performs better than state-of-the-art manual decomposition. Our approach is generic in nature and can decompose the standard transformers of the existing sub-polyhedra numerical abstract domains. We implemented our recipe and applied it to several practical numerical domains (e.g., Polyhedra, Octagon and Zones). Using a set of large Linux device drivers, we then evaluated the performance of our generated decomposed transformers vs. transformers obtained via state-of-the-art hand-tuned decomposition (Singh et al. 2015, 2017) showing that our approach leads to 2.4x (for Polyhedra) and 1.4x (for Octagon) speed-ups, on average. We believe this speed-up is due to our theorems (discussed next) which enable, in certain cases, finer decomposition of abstract elements than previously possible (indeed, we experimentally show that our permissible partitions are close to the finest partitions). Speedups compared to the original transformers without decomposition are orders of magnitude larger. Further, we decomposed the Zones domain using our approach (for which no previous decomposition exists) without changing the existing domain transformers. We obtain a speedup of 3x on average over non-decomposed implementation of the Zones domain. In summary, our recipe is generic in nature yet leads to state-of-the-art performance for classic abstract transformers.

5 DECOMPOSABLE TRANSFORMERS

When designing a decomposable transformer a key question is what partition is achievable for the output, given a partition of the input(s). In this section we define achievable partitions for all sub-polyhedra domains, focusing again on the conditional, assignment, meet, and join transformers. In the next section we will explain how to design the associated transformers either from scratch, or from an existing transformer (using our construction). Compared to (Halbwachs et al. 2003; Singh et al. 2015, 2017), we thus generalize decomposition to all sub-polyhedra domains. Further, in Section 6 we also show how to obtain finer partitions, including for Polyhedra and Octagon than (Singh et al. 2015, 2017), thus also obtaining significant speed-ups in our implementation.

In this section, we assume that inputs $\mathcal{I}, \mathcal{I}'$ for binary transformers are partitioned according to a common permissible partition $\overline{\pi}_{\text{common}}$. This partition can always be computed, that is, we have that $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ where $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ are any permissible partitions for $\mathcal{I}, \mathcal{I}'$, respectively. For the examples shown in this section, the permissible partitions for the inputs used and the obtained output are the finest.

5.1 Conditional

We consider conditional statements of the form $e \otimes c$ where $e := \sum_{i=1}^{n} a_i x_i$ with $a_i \in \mathbb{Z}, \otimes \in \{\leq, =\}$ and $c \in \mathbb{Q}, \mathbb{R}$ on an abstract element $\mathcal{I}$ with an associated permissible partition $\overline{\pi}_{\mathcal{I}}$ in domain $\mathcal{D}$. The conditional transformer computes the effect of adding the constraint $e \otimes c$ to $\mathcal{I}$. As discussed in

Section 2, a number of existing domains are not closed for the conditional transformer. Moreover, computing the best transformer is expensive in these domains and thus is usually approximated to strike a balance between precision and cost. The example below illustrates two sound conditional transformers on the same inputs, where the first transformer results in $\top$ partition and the second produces a decomposable output.

Example 5.1. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{polyhedra}} : (6, \mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{\{x_1 + x_2 \leq 0\}, \{x_3 + x_4 \leq 5\}\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\}.$$

For the conditional $x_5 + x_6 \leq 0$, a sound conditional transformer T_1 could produce output $\mathcal{I}'$ with partition $\top$:

$$\mathcal{I}' = \{\{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 5\}\} \text{ with } \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'} = \top.$$

Here, the output partition is $\top$ and thus T_1 is non-decomposable for this input. Another sound conditional transformer T_2 may return output $\mathcal{I}''$:

$$\mathcal{I}'' = \{\{x_1 + x_2 \leq 0\}, \{x_3 + x_4 \leq 5\}, \{x_5 + x_6 \leq 0\}\} \text{ with } \overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}.$$

In this case, $\pi_{\mathcal{I}''} \neq \top$ and thus, T_2 is decomposable for input $\mathcal{I}$.

Let $\mathcal{B}_{\text{cond}} = \{x_i \mid a_i \neq 0\}$ be the set of variables with non-zero coefficients in the constraint $\sum_{i=1}^{n} a_i x_i \otimes c$. The block $\mathcal{B}^*_{\text{cond}} = \bigcup_{\mathcal{X}_k \cap \mathcal{B}_{\text{cond}} \neq \emptyset} \mathcal{X}_k$ fuses all blocks $\mathcal{X}_k \in \overline{\pi}_{\mathcal{I}}$ that have non-empty intersection with $\mathcal{B}_{\text{cond}}$. We define the set $\mathcal{U}_c = \{x_i \mid x_i \notin \mathcal{B}^*_{\text{cond}}\}$ to contain the variables not in $\mathcal{B}^*_{\text{cond}}$.

Example 5.2. Consider $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and an element $\mathcal{I}$ in the Polyhedra domain with $\overline{\pi}_{\mathcal{I}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{x_6\}\}$. For the conditional $x_3 + x_6 \leq 0$, $\mathcal{B}_{\text{cond}} = \{x_3, x_6\}$ and $\mathcal{B}^*_{\text{cond}} = \{x_1, x_2, x_3, x_6\}$.

Next, we define a class of conditional transformers that provide a well-defined partitioned output. In Section 6 we then show that this class is not empty, i.e., the partitions are achievable in all sub-polyhedra domains.

Definition 5.1. A transformer T in $\mathcal{D}$ for the conditional statement $e \otimes c$ is in $[\![\text{cond}, \mathcal{D}]\!]$ iff for any element $\mathcal{I}$ with the permissible partition $\overline{\pi}_{\mathcal{I}}$ in $\mathcal{D}$, the output $\mathcal{I}'$ can be computed without creating non-redundant constraints between the variables from $\mathcal{B}^*_{\text{cond}}$ and the variables in $\mathcal{U}_c$.

While the construction of the output partition seems natural (and, as one can easily convince oneself, is satisfied by most standard transformers already in use), best transformers are not necessarily in this class due to constraints in the coefficient set $\mathcal{R}$ or the constant set C in the domain. We provide a counter example.

Example 5.3. We consider a fictive domain

$$\mathcal{X} = \{x_1, x_2\} \text{ and } \mathcal{L}_{\mathcal{X},\mathcal{D}} : (2, \mathbb{Z}^2, \{\leq, =\}, \{0, 1, 1.5\}).$$

We assume $\mathcal{I} = \{0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1\}$ with partition $\{\{x_1\}, \{x_2\}\}$ and the conditional $x_2 \leq 0.5$. In this case $\mathcal{B}^*_{\text{cond}} = \{x_2\}$. Using only constraints with variables in $\mathcal{B}^*_{\text{cond}}$ yields $\mathcal{I}' = \mathcal{I}$ as the most precise result since $0.5 \notin C$. However, the best transformer would produce $\mathcal{I}' = \mathcal{I} \cup \{x_1 + x_2 \leq 1.5\}$ or an equivalent abstract element. As a consequence, no best transformer in this domain is in $[\![\text{cond}, \mathcal{D}]\!]$.

5.2 Assignment

We consider linear assignments of the form $x_j := e$ on an abstract element $\mathcal{I}$ with an associated permissible partition $\overline{\pi}_{\mathcal{I}}$ in $\mathcal{D}$ where $e := \sum_{i=1}^{n} a_i x_i + c$ with $a_i \in \mathbb{Z}$ and $c \in \mathbb{Q}, \mathbb{R}$. An assignment is *invertible* if $a_j \neq 0$ (for example $x_1 := x_1 + x_2$). $\mathcal{I}_{x_j} \subseteq \mathcal{I}$ is the set of constraints with $a_j \neq 0$ in $\mathcal{I}$.

As discussed in Section 2, a number of existing domains are not closed for the assignment transformer. As for the conditional, the best assignment transformer is usually expensive for these domains and is overapproximated with a less precise one. The example below shows two sound approximations of the best assignment transformer on the same inputs, the first transformer results in the $\top$ partition whereas the second transformer keeps the output decomposed.

Example 5.4. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{Polyhedra}} : (6, \mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{\{x_1 + x_2 \leq 0\}, \{x_3 + x_4 \leq 5, x_5 - x_3 \leq 0\}\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4, x_5\}, \{x_6\}\}.$$

For the assignment $x_5 := -x_6$, a sound assignment transformer T_1 can produce the output $\mathcal{I}'$ with $\top$ partition:

$$\mathcal{I}' = \{\{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 5\}\} \text{ with } \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'} = \top.$$

Here, the output partition is $\top$. Thus, T_1 in non-decomposable w.r.t. the input $\mathcal{I}$. Another sound assignment transformer T_2 may return the output $\mathcal{I}''$:

$$\mathcal{I}'' = \{\{x_1 + x_2 \leq 0\}, \{x_3 + x_4 \leq 5\}, \{x_5 + x_6 = 0\}\} \text{ with } \overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}.$$

In this case, $\pi_{\mathcal{I}'} \neq \top$ and thus, T_2 is decomposable w.r.t. $\mathcal{I}$.

Let $\mathcal{B}_{\text{assign}} = \{x_i \mid a_i \neq 0\} \cup \{x_j\}$ be the set of variables containing x_j and all variables with non-zero coefficient in the linear expression $e := \sum_{i=1}^{n} a_i x_i + c$. The block $\mathcal{B}^*_{\text{assign}} = \bigcup_{\mathcal{X}_k \cap \mathcal{B}_{\text{assign}} \neq \emptyset} \mathcal{X}_k$ fuses all blocks $\mathcal{X}_k \in \overline{\pi}_{\mathcal{I}}$ having non-empty intersection with $\mathcal{B}_{\text{assign}}$. We define the set $\mathcal{U}_a = \{x_i \mid x_i \notin \mathcal{B}^*_{\text{assign}}\}$ to contain variables not in $\mathcal{B}^*_{\text{assign}}$.

Example 5.5. Consider $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and an element $\mathcal{I}$ in the Polyhedra domain with $\overline{\pi}_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}$. For the assignment $x_3 := x_1 + x_2$, $\mathcal{B}_{\text{assign}} = \{x_1, x_2, x_3\}$ and $\mathcal{B}^*_{\text{assign}} = \{x_1, x_2, x_3, x_4\}$.

Generic transformer for invertible assignment. The invertible assignment transformer removes all constraints in $\mathcal{I}_{x_j}$ from $\mathcal{I}$. It then computes a set of constraints $\mathcal{I}_{\text{inv}}$ by substituting $(x_j - \sum_{i \neq j} a_i x_i - c)/a_j$ for x_j in all constraints in $\mathcal{I}_{x_j}$. Finally, it adds a set of representable constraints $\mathcal{I}'_{\text{inv}}$ capturing the effect of the addition of $\mathcal{I}_{\text{inv}}$ to $\mathcal{I} \setminus \mathcal{I}_{x_j}$ using the conditional transformer.

Generic transformer for non-invertible assignment. The non-invertible assignment transformer removes all constraints in $\mathcal{I}_{x_j}$ from $\mathcal{I}$. Next, it computes a set of constraints $\mathcal{I}_{\text{non-inv}}$ by *projecting* x_j from all constraints in $\mathcal{I}_{x_j}$ using variable elimination. Finally, it adds a set of representable constraints $\mathcal{I}'_{\text{non-inv}}$ capturing the effect of the addition of $\mathcal{I}_{\text{non-inv}} \cup \{x_j - e = 0\}$ to $\mathcal{I} \setminus \mathcal{I}_{x_j}$ using the conditional transformer.

We use the above constructions to define a class $[\![\text{assign}, \mathcal{D}]\!]$ of decomposed assignment transformers for the statement $x_j := e$ based on $\mathcal{B}^*_{\text{assign}}$ in $\mathcal{D}$.

Definition 5.2. An assignment transformer T in $\mathcal{D}$ for the statement $x_j := e$ is in $[\![\text{assign}, \mathcal{D}]\!]$ iff for any element $\mathcal{I}$ with the permissible partition $\overline{\pi}_{\mathcal{I}}$ in $\mathcal{D}$, the output $\mathcal{I}'$ can be computed without creating non-redundant constraints between the variables in $\mathcal{B}^*_{\text{assign}}$ and the variables in $\mathcal{U}_a$.

An example of a domain in which no best transformer $\in [\![\text{assign}, \mathcal{D}]\!]$ can be constructed as for the conditional.

5.3 Meet (⊓)

As discussed in Section 2, all existing domains are closed for the meet (⊓) transformer. Moreover, they always implement the best transformer as it is simply the union of the inputs $\mathcal{I}, \mathcal{I}'$ making it easy to compute. An approximation of the best transformer may result in loss of the lattice property $\mathcal{I} \sqcap \mathcal{I}' \sqsubseteq \mathcal{I}$ and $\mathcal{I} \sqcap \mathcal{I}' \sqsubseteq \mathcal{I}'$. An arbitrary approximation can again result in the $\top$ partition.

Definition 5.3. A meet transformer T in $\mathcal{D}$ is in $[\![\sqcap, \mathcal{D}]\!]$ iff for input elements $\mathcal{I}, \mathcal{I}'$ with the common permissible partition $\overline{\pi}_{\text{common}}$ the output $T(\mathcal{I}, \mathcal{I}')$ can be computed from $\mathcal{I}, \mathcal{I}'$ without creating non-redundant constraints between the variables in different blocks of $\overline{\pi}_{\text{common}}$.

In this case, the best transformer is in $[\![\sqcap, \mathcal{D}]\!]$ since it is simply obtained as $\mathcal{I} \cup \mathcal{I}'$, which is representable in the domain and does not create non-redundant constraints between the variables in different blocks of $\overline{\pi}_{\text{common}}$.

5.4 Join (⊔)

As discussed in Section 2, none of the existing domains are closed for the join (⊔) transformer. The join transformer approximates the union of $\mathcal{I}$ and $\mathcal{I}'$ in $\mathcal{D}$ and is usually the most expensive transformer in $\mathcal{D}$ and thus approximated. As with other transformers, an arbitrary approximation can result in the $\top$ partition for all equivalent outputs. The example below shows two sound join transformers in the Zones domain. The first transformer produces the $\top$ partition whereas the second one preserves it.

Example 5.6. Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{zones}} : (6, \{1, 0\} \times \{0, -1\} \times \{0\}^4, \{\leq, =\}, \mathbb{R}), \\
\mathcal{I} &= \{\{x_1 = 1\}, \{x_2 = 2\}, \{x_3 \leq 3\}, \{x_4 = 4\}, \{x_5 = 0\}, \{x_6 = 0\}\} \text{ and} \\
\mathcal{I}' &= \{\{x_1 = 1\}, \{x_2 = 2\}, \{x_3 \leq 3\}, \{x_4 = 4\}, \{x_5 = 1\}, \{x_6 = 1\}\} \text{ with} \\
\overline{\pi}_{\mathcal{I}} &= \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}} = \bot.
\end{aligned}$$

One sound transformer T_1 for the join transformer could produce the output $\mathcal{I}''$ with $\top$ partition:

$$\mathcal{I}'' = \{\{x_2 - x_1 \leq 1, x_1 - x_5 \leq 1, x_3 - x_2 \leq 1, x_4 - x_3 \leq 1, x_5 = x_6\}\}$$
$$\text{with } \overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''} = \top.$$

Thus T_1 is not decomposable w.r.t. the inputs $\mathcal{I}, \mathcal{I}'$. Another sound transformer T_2 may return the output $\mathcal{I}'''$:

$$\mathcal{I}''' = \{\{x_1 = 1\}, \{x_2 = 2\}, \{x_3 \leq 3\}, \{x_4 = 4\}, \{-x_5 \leq 0, x_5 \leq 1\}, \{-x_6 \leq 0, x_6 \leq 1\}\}$$
$$\text{with } \overline{\pi}_{\mathcal{I}'''} = \pi_{\mathcal{I}'''} = \bot.$$

In this case $\pi_{\mathcal{I}'''} \neq \top$ and thus T_2 is decomposable w.r.t the inputs $\mathcal{I}, \mathcal{I}'$.

Let $\mathcal{E} = \{\mathcal{X}_k \mid \mathcal{X}_k \in \overline{\pi}_{\text{common}}, \mathcal{I}_k = \mathcal{I}'_k\}$ be the set of blocks such that the corresponding factors $\mathcal{I}_k, \mathcal{I}'_k$ are equal. Let $\mathcal{N} = \bigcup\{\mathcal{X}_k \mid \mathcal{X}_k \in \overline{\pi}_{\text{common}}, \mathcal{I}_k \neq \mathcal{I}'_k\}$ be the union of all remaining blocks.

Definition 5.4. A join transformer T in $\mathcal{D}$ is in $[\![\sqcup, \mathcal{D}]\!]$ iff for input elements $\mathcal{I}, \mathcal{I}'$ with the common permissible partition $\overline{\pi}_{\text{common}}$ the output $T(\mathcal{I}, \mathcal{I}')$ can be computed from $\mathcal{I}, \mathcal{I}'$ by creating non-redundant constraints between only the variables in the set $\mathcal{N}$.

In example 5.6, we have $\mathcal{E} = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}\}$ and $\mathcal{N} = \{x_5, x_6\}$. $T_1 \notin [\![\sqcup, \mathcal{D}]\!]$ as T_1 creates a non-redundant constraint between x_1 and x_2 which are in different blocks of $\mathcal{E}$ whereas $T_2 \in [\![\sqcup, \mathcal{D}]\!]$.

5.5 Widening ($\triangledown$)

The widening transformer ($\triangledown$) is applied during analysis to accelerate convergence towards a fixpoint. It is a binary transformer and guarantees that: (i) the output $\mathcal{I}'' \sqsupseteq \mathcal{I}$, (ii) $\mathcal{I}'' \sqsupseteq \mathcal{I}'$, and (iii) the analysis terminates after a finite number of steps. The best widening transformer does not exist for any numerical domain. In theory, it may be possible to design arbitrary widening transformers that always result in the $\top$ partition. In practice, the standard widening transformers are of two types:

Syntactic. For syntactic widening, the set of constraints in the output element $\mathcal{I}''$ is $\subseteq \mathcal{I}$. A constraint $\iota := \sum_{i=1}^{n} a_i x_i \leq c \in \mathcal{I}$ is in the output $\mathcal{I}''$ iff there is a constraint $\iota' := \sum_{i=1}^{n} a_i x_i \leq c' \in \mathcal{I}'$ with the same linear expression and $c' \leq c$.

Semantic. The semantic widening (Cousot et al. 2005) requires the input $\mathcal{I}$ to be minimal. The set of constraint in the output $\mathcal{I}''$ is $\subseteq \mathcal{I} \cup \mathcal{I}'$. $\mathcal{I}''$ contains the constraints from $\mathcal{I}$ that are satisfied by $\mathcal{I}'$ and the constraints ι' from $\mathcal{I}'$ that are mutually redundant with a constraint ι in $\mathcal{I}$.

Both these transformers are decomposable in practice. The following example illustrates the semantic and the syntactic widening on the Octagon domain.

Example 5.7. Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\text{octagon}} : (4, \mathbb{U}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{I}), \\
\mathcal{I} &= \{\{x_1 - x_2 \leq 0, x_2 \leq 0\}, \{x_3 \leq 0\}, \{x_4 \leq 1\}\}, \mathcal{I}' = \{\{x_1 \leq 0\}, \{x_3 + x_4 \leq 2\}\}, \text{ with} \\
\overline{\pi}_{\mathcal{I}} &= \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\} \text{ and } \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'} = \{\{x_1, x_2\}, \{x_3, x_4\}\}.
\end{aligned}$$

The semantic widening transformer T_1 yields:

$$\mathcal{I}'' = \{\{x_1 \leq 0\}\} \text{ with } \overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''} = \bot.$$

On the other hand, the syntactic widening transformer T_2 yields:

$$\mathcal{I}''' = \emptyset \text{ with } \overline{\pi}_{\mathcal{I}'''} = \pi_{\mathcal{I}'''} = \bot.$$

For both T_2 and T_2 the output partition is $\neq \top$ and thus both are decomposable w.r.t. the inputs $\mathcal{I}, \mathcal{I}'$.

We define the class $[\![\triangledown, \mathcal{D}]\!]$ of widening transformers in $\mathcal{D}$ to contain the standard transformers.

Definition 5.5. A widening transformer T is in $[\![\triangledown, \mathcal{D}]\!]$ iff for input elements $\mathcal{I}, \mathcal{I}'$ with the common permissible partition $\overline{\pi}_{\text{common}}$ the output $T(\mathcal{I}, \mathcal{I}')$ can be computed from $\mathcal{I}, \mathcal{I}'$ without creating non-redundant constraints between the variables in different blocks of $\overline{\pi}_{\text{common}}$.

In example 5.7, both T_1 and T_2 are in $[\![\triangledown, \mathcal{D}]\!]$. We write $T_\triangledown$ for a transformer in $[\![\triangledown, \mathcal{D}]\!]$. It can be shown that the standard transformer $T_\triangledown^{\text{stan}}$ in existing domains is in $[\![\triangledown, \mathcal{D}]\!]$.

6 DECOMPOSING DOMAIN TRANSFORMERS

In this section, we show a construction which takes as input a transformer (e.g., for a conditional or an assignment) in a given domain $\mathcal{D}$ and produces a (decomposed) transformer in the classes defined in Section 5, i.e., it guarantees an upper bound for the partition of the output. The decomposed transformer is obtained as already sketched informally in Section 4, i.e., the given transformer is applied on smaller abstract elements and then the result assembled, which reduces complexity and thus improves analysis performance.

Algorithm 1 Decomposed conditional transformer

function CONDITIONAL($\mathcal{I}, \overline{\pi}_{\mathcal{I}}$, stmt, T_{cond})
  Parameters:
    $\mathcal{I} \leftarrow \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_p\}$
    $\overline{\pi}_{\mathcal{I}} \leftarrow \{\mathcal{X}_{\mathcal{I}_1}, \mathcal{X}_{\mathcal{I}_2}, \ldots, \mathcal{X}_{\mathcal{I}_p}\}$
    stmt $\leftarrow e \otimes c$
    $T_{\text{cond}} \leftarrow$ conditional transformer
  $\mathcal{B}^*_{\text{cond}}$:= extract_block(stmt, $\overline{\pi}_{\mathcal{I}}$)
  $\mathcal{U}_{\text{c}} := \{x_i \mid x_i \notin \mathcal{B}^*_{\text{cond}}\}$
  $\pi_{\text{cond}} := \{\mathcal{B}^*_{\text{cond}}, \{u_1\}, \ldots, \{u_r\}\}, u_i \in \mathcal{U}_{\text{c}}$
  $\overline{\pi} := \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{cond}} := \{\mathcal{X}_{\mathcal{I}^r_1}, \mathcal{X}_{\mathcal{I}^r_2}, \ldots, \mathcal{X}_{\mathcal{I}^r_l}\}$
  $\mathcal{I}^r$:= refactor($\mathcal{I}, \overline{\pi}_{\mathcal{I}}, \overline{\pi}$)
  $\mathcal{I}' := \emptyset$
  for $k \in \{1, 2, \ldots, l\}$ **do**
    if $\mathcal{X}_{\mathcal{I}^r_k} = \mathcal{B}^*_{\text{cond}}$ **then**
      $\mathcal{I}'$.add($T_{\text{cond}}(\mathcal{I}^r_k)$)
    else
      $\mathcal{I}'$.add($\mathcal{I}^r_k$)
  $\overline{\pi}_{\mathcal{I}'} := \overline{\pi}$

Soundness. The main task is to show that for a given transfomer T, our obtained decomposed T' is sound. As usual, this is the case iff for any element $\mathcal{I} \in \mathcal{D}$, $\gamma(T^{\text{best}}(\mathcal{I})) \subseteq \gamma(T'(\mathcal{I}))$ (the criterion is naturally extended to transformers with multiple arguments). Note that in general it can happen that $\gamma(T(\mathcal{I})) \subset \gamma(T'(\mathcal{I}))$ or $\gamma(T(\mathcal{I})) \supset \gamma(T'(\mathcal{I}))$, i.e., the decomposed transformer T' may have better or worse precision with respect to the original non-decomposed transformer T.

Quality of partition. Our construction guarantees that the output partitions obey the definitions in Section 5. In the process, we show how to obtain refinements for the output partition that do not create significant overhead while yielding significant performance gains.

In this section, we use the notation $\beta_{\mathcal{B}}(\mathcal{I})$ to denote the projection of $\mathcal{I}$ defined over $\mathcal{X}$ to a subset $\mathcal{B}$ of $\mathcal{X}$.

6.1 Conditional

Algorithm 1 shows a construction for decomposing a given conditional transformer T_{cond}. Given an input element $\mathcal{I}$ with a permissible partition $\overline{\pi}_{\mathcal{I}}$ in domain $\mathcal{D}$, the algorithm first extracts the block $\mathcal{B}^*_{\text{cond}}$ based on the conditional statement and the permissible partition $\overline{\pi}_{\mathcal{I}}$. It then computes the set of variables $\mathcal{U}_{\text{c}}$ that are not in $\mathcal{B}^*_{\text{cond}}$ followed by a partition $\pi_{\text{cond}} = \{\mathcal{B}^*_{\text{cond}}, \{u_1\}, \ldots, \{u_r\}\}\}, u_i \in \mathcal{U}_{\text{c}}$ corresponding to the conditional statement $e \otimes c$. The input $\mathcal{I}$ is refactored with respect to the partition $\overline{\pi} = \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{cond}}$ producing $\mathcal{I}^r$. The transformer T_{cond} is applied only on the factor $\mathcal{I}^r_{\text{cond}}$ of $\mathcal{I}^r$ corresponding to the block $\mathcal{B}^*_{\text{cond}}$. By applying T_{cond} to $\mathcal{I}^r_{\text{cond}}$ only, we reduce complexity and thus increase performance. The following example illustrates the decomposition of a conditional transformer in the TVPI domain using Algorithm 1.

Example 6.1. Let

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5\}, \mathcal{L}_{\mathcal{X},\text{tvpi}} : (\mathbb{Z}^2 \times \{0\}^3, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{\{x_1 \leq x_2\}, \{x_3 + x_4 \leq 5\}, \{x_5 = 7\}\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}\}.$$

Consider the conditional statement $2x_4 + x_5 \leq 8$ with $\mathcal{B}_{\text{cond}} = \{x_4, x_5\}$. Algorithm 1 computes $\mathcal{B}^*_{\text{cond}} = \{x_3, x_4, x_5\}$, $\pi_{\text{cond}} = \{\{x_1\}, \{x_2\}, \{x_3, x_4, x_5\}\}$ and $\overline{\pi} = \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{cond}} = \{\{x_1, x_2\}, \{x_3, x_4, x_5\}\}$. It then refactors $\mathcal{I}$ with respect to $\overline{\pi}$ producing $\mathcal{I}^r$:

$$\mathcal{I}^r = \{\{x_1 \leq x_2\}, \{x_3 + x_4 \leq 5, x_5 = 7\}\}.$$

Finally, T_{cond} is applied only on $\mathcal{I}^r_2$:

$$\mathcal{I}' = \{\mathcal{I}^r_1, T_{\text{cond}}(\mathcal{I}^r_2)\} = \{\{x_1 \leq x_2\}, \{x_3 + x_4 \leq 5, x_5 = 7, 2x_4 + x_5 \leq 8\}\}.$$

By construction, the decomposed transformer in Algorithm 1 is in $[\![\text{cond}, \mathcal{D}]\!]$; it remains to show soundness.

Theorem 6.1. *Let T_{cond} be a conditional transformer for the statement $e \otimes c$. Then the associated decomposed transformer T^{D}_{cond} (Algorithm 1) is sound, i.e., $\gamma(T^{best}_{cond}(\mathcal{I})) \subseteq \gamma(T^{D}_{cond}(\mathcal{I}))$ for all $\mathcal{I}$.*

Proof. By construction $\mathcal{I} = \mathcal{I}^r$. Algorithm 1 applies T_{cond} on $\mathcal{I}^r_{\text{cond}}$ defined over $\mathcal{B}^*_{\text{cond}}$ only. Thus, we can write $T^{D}_{\text{cond}}(\mathcal{I}^r) = T_{\text{cond}}(\beta_{\mathcal{B}^*_{\text{cond}}}(\mathcal{I}^r_{\text{cond}})) \cup \mathcal{I}^r_{\mathcal{U}_c}$ where $\mathcal{I}^r_{\mathcal{U}_c}$ contains the set of constraints in $\mathcal{I}^r$ that are not in $\mathcal{I}^r_{\text{cond}}$.

$$\begin{aligned}
T^{\text{best}}_{\text{cond}}(\mathcal{I}^r) &= T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\text{cond}} \cup \mathcal{I}^r_{\mathcal{U}_c}) \\
&\sqsubseteq \beta_{\mathcal{B}^*_{\text{cond}}}(T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\text{cond}})) \times \beta_{\mathcal{U}_c}(T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\mathcal{U}_c})) \\
&\sqsubseteq \beta_{\mathcal{B}^*_{\text{cond}}}(T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\text{cond}})) \times \beta_{\mathcal{U}_c}(\mathcal{I}^r_{\mathcal{U}_c}) && \text{(By definition } T^{\text{best}}_{\text{cond}}(\mathcal{I}) \sqsubseteq \mathcal{I}) \\
&= T^{\text{best}}_{\text{cond}}(\beta_{\mathcal{B}^*_{\text{cond}}}(\mathcal{I}^r_{\text{cond}}))) \cup \mathcal{I}^r_{\mathcal{U}_c} \\
&\sqsubseteq T_{\text{cond}}(\beta_{\mathcal{B}^*_{\text{cond}}}(\mathcal{I}^r_{\text{cond}})) \cup \mathcal{I}^r_{\mathcal{U}_c} \\
&= T^{D}_{\text{cond}}(\mathcal{I}^r).
\end{aligned}$$

Since γ is monotone, we have $\gamma(T^{\text{best}}_{\text{cond}}(\mathcal{I})) \subseteq \gamma(T^{D}_{\text{cond}}(\mathcal{I}))$ and thus the theorem holds. □

The definition of $\mathcal{B}^*_{\text{cond}}$ guarantees that $\beta_{\mathcal{B}^*_{\text{cond}}}(T_{\text{cond}}(\mathcal{I}^r_{\mathcal{B}^*_{\text{cond}}})) = T_{\text{cond}}(\beta_{\mathcal{B}^*_{\text{cond}}}(\mathcal{I}^r_{\text{cond}}))$ holds for any conditional transformer T_{cond} in $\mathcal{D}$. The proof of Theorem 6.1 requires that $\beta_{\mathcal{B}}(T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\mathcal{B}})) \sqsubseteq T^{\text{best}}_{\text{cond}}(\beta_{\mathcal{B}}(\mathcal{I}^r_{\mathcal{B}})))$ which may not hold for any arbitrary $\mathcal{B}$. The following example illustrates this for a conditional transformer in the Octagon domain.

Example 6.2. Consider

$$\mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X},\text{octagon}} : (3, \mathbb{U}^3, \{\leq, =\}, \mathbb{R}), \mathcal{B} = \{x_1, x_2\}, \mathcal{I}^r_{\mathcal{B}} = \{x_1 \leq 0, x_2 \leq 0\}.$$

The best conditional transformer for the statement $x_3 \leq 0$ which adds the constraint $x_3 \leq 0$ is:

$$\beta_{\mathcal{B}}(T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\mathcal{B}})) = \{x_1 \leq 0, x_2 \leq 0\} \sqsupset T^{\text{best}}_{\text{cond}}(\beta_{\mathcal{B}}(\mathcal{I}^r_{\mathcal{B}}))) = \{x_1 \leq 0, x_2 \leq 0, x_3 \leq 0\}.$$

In general, the block $\mathcal{B}$ should contain $\mathcal{B}_{\text{cond}}$ to ensure soundness. T_{cond} in Algorithm 1 creates constraints between the variables in $\mathcal{B}^*_{\text{cond}}$ only. The partition $\overline{\pi}_{\mathcal{I}'} = \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{cond}}$ contains $\mathcal{B}^*_{\text{cond}}$ as a block and is thus permissible for the output $\mathcal{I}'$. Since we do not know the exact constraints in the output $\mathcal{I}'$, $\overline{\pi}_{\mathcal{I}'} \neq \pi_{\mathcal{I}'}$ in general even if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$. The following corollary provides conditions when the output partition $\overline{\pi}_{\mathcal{I}'}$ computed by Algorithm 1 is finest, i.e., $\overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$.

Corollary 6.2. *$\overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$, if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$ and $\mathcal{I}' = \mathcal{I} \cup \{e \otimes c\}$.*

6.2 Assignment

Algorithm 2 shows our construction of a decomposed transformer for a given assignment transformer T_{assign} for an input element $\mathcal{I}$ with the associated permissible partition $\overline{\pi}_{\mathcal{I}}$ in domain $\mathcal{D}$. The algorithm extracts the block $\mathcal{B}^*_{\text{assign}}$ based on the assignment statement and the input partition $\overline{\pi}_{\mathcal{I}}$. Next, It computes the set of variables $\mathcal{U}_{\text{a}}$ that are not in $\mathcal{B}^*_{\text{assign}}$ followed by the partition $\pi_{\text{assign}} = \{\mathcal{B}^*_{\text{assign}}, \{u_1\}, \ldots, \{u_r\}\}\}, u_i \in \mathcal{U}_{\text{a}}$ corresponding to the assignment statement $x_j := e$. The input $\mathcal{I}$ is refactored with respect to the partition $\overline{\pi} = \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{assign}}$ producing $\mathcal{I}^r$. The algorithm applies the transformer T_{assign} only on the factor $\mathcal{I}^r_{\text{assign}}$ of $\mathcal{I}^r$ corresponding to the block $\mathcal{B}^*_{\text{assign}}$.

Algorithm 2 Decomposed assignment transformer

function Assignment($\mathcal{I}, \overline{\pi}_{\mathcal{I}}$, stmt, T_{assign})
    Parameters:
        $\mathcal{I} \leftarrow \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_p\}$
        $\overline{\pi}_{\mathcal{I}} \leftarrow \{\mathcal{X}_{\mathcal{I}_1}, \mathcal{X}_{\mathcal{I}_2}, \ldots, \mathcal{X}_{\mathcal{I}_p}\}$
        stmt $\leftarrow x_j := e$
        $T_{\text{assign}} \leftarrow$ Assignment transformer
    $\mathcal{B}^*_{\text{assign}} :=$ extract_block(stmt, $\overline{\pi}_{\mathcal{I}}$)
    $\mathcal{U}_{\text{a}} := \{x_i \mid x_i \notin \mathcal{B}^*_{\text{assign}}\}$
    $\pi_{\text{assign}} := \{\mathcal{B}^*_{\text{assign}}, \{u_1\}, \ldots, \{u_r\}\}, u_i \in \mathcal{U}_{\text{a}}$
    $\overline{\pi} := \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{assign}} := \{\mathcal{X}_{\mathcal{I}^r_1}, \mathcal{X}_{\mathcal{I}^r_2}, \ldots, \mathcal{X}_{\mathcal{I}^r_l}\}$
    $\mathcal{I}^r :=$ refactor$(\mathcal{I}, \overline{\pi}_{\mathcal{I}}, \overline{\pi})$
    $\mathcal{I}' := \emptyset$
    for $k \in \{1, 2, \ldots, l\}$ **do**
        if $\mathcal{X}_{\mathcal{I}^r_k} = \mathcal{B}^*_{\text{assign}}$ **then**
            $\mathcal{I}'$.add$(T_{\text{assign}}(\mathcal{I}^r_k))$
        else
            $\mathcal{I}'$.add$(\mathcal{I}^r_k)$
    $\overline{\pi}_{\mathcal{I}'} := \overline{\pi}$

Algorithm 2 applies T_{assign} on the factor $\mathcal{I}^r_{\text{assign}}$ only which reduces its complexity. The following example illustrates the decomposition of an assignment transformer for the Polyhedra domain using Algorithm 2.

Example 6.3. Let:

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{polyhedra}} : (\mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{\{x_1 \leq x_2, x_2 + x_3 \leq 5\}, \{x_4 - x_5 \leq 3\}, \{x_6 \leq 7\}\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{x_6\}\}.$$

Consider an invertible assignment $x_4 := x_4 + x_6$ with $\mathcal{B}_{\text{assign}} = \{x_4, x_6\}$. Algorithm 2 computes $\mathcal{B}^*_{\text{assign}} = \{x_4, x_5, x_6\}$, $\pi_{\text{assign}} = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4, x_5, x_6\}\}$ and $\overline{\pi} = \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{assign}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$. It then refactors $\mathcal{I}$ with respect to $\overline{\pi}$ producing $\mathcal{I}^r$:

$$\mathcal{I}^r = \{\{x_1 \leq x_2, x_2 + x_3 \leq 5\}, \{x_4 - x_5 \leq 3, x_6 \leq 7\}\}.$$

It then applies T_{assign} on $\mathcal{I}^r_2$ only to produce the output $\mathcal{I}'$:

$$\mathcal{I}' = \{\mathcal{I}^r_1, T_{\text{assign}}(\mathcal{I}^r_2)\} = \{\{x_1 \leq x_2, x_2 + x_3 \leq 5\}, \{x_4 - x_5 - x_6 \leq 3, x_6 \leq 7\}\}.$$

By construction the decomposed transformer is in $[\![\text{assign}, \mathcal{D}]\!]$; it remains to show soundness.

Theorem 6.3. *Let T_{assign} be an assignment transformer for the assignment statement $x_j := e$. Then the associated decomposed transformer T^D_{assign} (Algorithm 2) is sound, i.e., $\gamma(T^{best}_{assign}(\mathcal{I})) \subseteq \gamma(T^D_{assign}(\mathcal{I}))$ for all $\mathcal{I}$.*

Proof. By construction $\mathcal{I} = \mathcal{I}^r$. Algorithm 2 applies T_{assign} on $\mathcal{I}^r_{\text{assign}}$ defined over $\mathcal{B}^*_{\text{assign}}$ only. Thus, we can write $T^D_{\text{assign}}(\mathcal{I}^r) = T_{\text{assign}}(\beta_{\mathcal{B}^*_{\text{assign}}}(\mathcal{I}^r_{\text{assign}})) \cup \mathcal{I}^r_{\mathcal{U}_a}$ where $\mathcal{I}^r_{\mathcal{U}_a}$ contains the set of constraints in $\mathcal{I}^r$ that are not in $\mathcal{I}^r_{\text{assign}}$. Since $x_j \in \mathcal{B}^*_{\text{assign}}$, it follows that $\mathcal{I}^r \setminus \mathcal{I}_{x_j} = (\mathcal{I}^r_{\text{assign}} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}^r_{\mathcal{U}_a}$. If the assignment statement is non-invertible, all constraints in $\mathcal{I}_{\text{non-inv}}$ created by eliminating x_j from $\mathcal{I}_{x_j}$ can be obtained by eliminating x_j from $\mathcal{I}^r_{\text{assign}}$ only. Similarly for the invertible assignment, all constraints in $\mathcal{I}_{\text{inv}}$ created by substituting $(x_j - \sum_{i \neq j} a_i x_i - c)/a_j$ for x_j in all constraints in $\mathcal{I}_{x_j}$ can be obtained by substituting for x_j in $\mathcal{I}^r_{\text{assign}}$. T_{assign} adds each constraint $\iota \in \mathcal{I}_{\text{non-inv}} \cup \{x_j - e = 0\}$ or $\iota' \in \mathcal{I}_{\text{inv}}$ to $\mathcal{I}^r \setminus \mathcal{I}_{x_j}$ through the conditional transformer. Each ι or ι' contains variables from $\mathcal{B}^*_{\text{assign}}$ only. By the definition of $\mathcal{B}^*_{\text{assign}}$ and the soundness of Theorem 6.1, each ι can be soundly added to $\mathcal{I}^r \setminus \mathcal{I}_{x_j}$ by applying the conditional transformer on $\mathcal{I}^r_{\text{assign}}$. Thus, $\gamma(T^{\text{best}}_{\text{assign}}(\mathcal{I})) \subseteq \gamma(T^D_{\text{assign}}(\mathcal{I}))$ holds. □

Algorithm 3 Decomposed meet transformer

function MEET($\mathcal{I}, \mathcal{I}', \overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}, T_{\sqcap}$)
  Parameters:
    $\mathcal{I} \leftarrow \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_p\}$
    $\mathcal{I}' \leftarrow \{\mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_q\}$
    $\overline{\pi}_{\mathcal{I}} \leftarrow \{\mathcal{X}_{\mathcal{I}_1}, \mathcal{X}_{\mathcal{I}_2}, \ldots, \mathcal{X}_{\mathcal{I}_p}\}$
    $\overline{\pi}_{\mathcal{I}'} \leftarrow \{\mathcal{X}_{\mathcal{I}'_1}, \mathcal{X}_{\mathcal{I}'_2}, \ldots, \mathcal{X}_{\mathcal{I}'_q}\}$
    $T_{\sqcap} \leftarrow$ meet transformer
  $\mathcal{I}'' := \emptyset$
  $\overline{\pi}_{\text{common}} := \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$
  $\mathcal{I}^r :=$ refactor($\mathcal{I}, \overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\text{common}}$)
  $\mathcal{I}'^r :=$ refactor($\mathcal{I}', \overline{\pi}_{\mathcal{I}'}, \overline{\pi}_{\text{common}}$)
  for $k \in \{1, 2, \ldots, l\}$ **do**
    $\mathcal{I}''$.add($T_{\sqcap}(\mathcal{I}^r_k, \mathcal{I}'^r_k)$)
  $\overline{\pi}_{\mathcal{I}''} := \overline{\pi}_{\text{common}}$

It is easy to see that it is unsound to apply T_{assign} on a factor corresponding to a block that does not contain x_j. T_{assign} in Algorithm 2 creates constraints between the variables in $\mathcal{B}^*_{\text{assign}}$ only. Thus, $\overline{\pi}_{\mathcal{I}'} = \overline{\pi}_{\mathcal{I}} \sqcup \pi_{\text{assign}}$ contains $\mathcal{B}^*_{\text{assign}}$ as a block and is permissible for the output $\mathcal{I}'$.

Refinement. Let $\mathcal{B}_{x_j}$ be the block containing x_j in $\overline{\pi}_{\mathcal{I}}$. If $\mathcal{B}_{x_j} \cap (\mathcal{B}_{\text{assign}} \setminus \{x_j\}) = \emptyset$ and the assignment statement is non-invertible (e.g., $x_1 := x_2 + x_3$), we can modify Algorithm 2 to work on finer partitions. We define the block $\mathcal{B}'^*_{\text{assign}} = \mathcal{B}^*_{\text{assign}} \setminus (\mathcal{B}_{x_j} \setminus \{x_j\})$ to contain all variables from $\mathcal{B}^*_{\text{assign}}$ except the variables in $\mathcal{B}_{x_j} \setminus \{x_j\}$. Let $\mathcal{U}'_{\text{a}} = \{x_i \mid x_i \notin \mathcal{B}'^*_{\text{assign}}\}$ be the set of variables not in $\mathcal{B}'^*_{\text{assign}}$ and $\pi'_{\text{assign}} = \{\mathcal{B}'^*_{\text{assign}}, \{u'_1\}, \ldots, \{u'_r\}\}$ where $u'_i \in \mathcal{U}'_a$ is the partition corresponding to the non-invertible assignment with $\mathcal{B}_{x_j} \cap (\mathcal{B}_{\text{assign}} \setminus \{x_j\}) = \emptyset$. We compute the partition $\overline{\pi}' = (\overline{\pi}_{\mathcal{I}} \sqcap \{\mathcal{X} \setminus \{x_j\}, \{x_j\}\}) \sqcup \pi'_{\text{assign}}$ which is finer than $\overline{\pi}$ in Algorithm 2. $\overline{\pi}'$ splits the block $\mathcal{B}^*_{\text{assign}} \in \overline{\pi}$ into two blocks $\mathcal{B}_{x_j} \setminus \{x_j\}$ and $\mathcal{B}'^*_{\text{assign}}$. $\mathcal{I}_{x_j}$ and $\mathcal{I}_{\text{non-inv}}$ is computed by applying T_{assign} on $\mathcal{I}^r_{\text{assign}}$ as before. $\mathcal{I}^r_{\text{assign}}$ is then split into two factors: $\mathcal{I}^r_{\mathcal{B}_{x_j}}$ and $\mathcal{I}^r_{\text{assign}'}$, corresponding to the blocks $\mathcal{B}_{x_j}$ and $\mathcal{B}'^*_{\text{assign}}$ respectively. Then, the constraints in $\mathcal{I}_{\text{non-inv}}$ contain variables from $\mathcal{B}_{x_j} \setminus \{x_j\}$ and can be added to $\mathcal{I} \setminus \mathcal{I}_{x_j}$ by applying T_{assign} on the factor $\mathcal{I}^r_{\mathcal{B}_{x_j}}$ while the constraint $x_j - e = 0$ can be added by applying T_{assign} on the factor $\mathcal{I}^r_{\text{assign}'}$.

The modified algorithm applies T_{assign} on smaller factors and is sound by construction. $\overline{\pi}'_{\mathcal{I}'} = \overline{\pi}'$ is permissible for $\mathcal{I}'$. The following corollary provides conditions for checking when $\overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ after applying T_{assign} for the invertible assignment statement.

Corollary 6.4. *For the invertible assignment statement $x_j := e$, $\overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$ and $\mathcal{I} = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}_{inv}$.*

The following corollary defines conditions for checking when the output partition $\overline{\pi}_{\mathcal{I}'}$ or the refinement $\overline{\pi}'_{\mathcal{I}'}$ after applying T_{assign} for the non-invertible assignment statement on $\mathcal{I}$ is finest.

Corollary 6.5. *For the non-invertible assignment statement $x_j := e$ with $\mathcal{B}_{x_j} \cap (\mathcal{B}_{assign} \setminus \{x_j\}) = \emptyset$, $\overline{\pi}'_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$, $\mathcal{I}' = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup (\mathcal{I}_{non\text{-}inv} \cup \{x_j - e = 0\})$. If $\mathcal{B}_{x_j} \cap (\mathcal{B}_{assign} \setminus \{x_j\}) \neq \emptyset$ then $\overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ if the same conditions on $\mathcal{I}'$ and $\overline{\pi}_{\mathcal{I}}$ are satisfied.*

6.3 Meet ($\sqcap$)

Algorithm 3 shows our construction of a decomposed transformer for a given meet transformer $T_{\sqcap} \in [\![\sqcap, \mathcal{D}]\!]$ on input elements $\mathcal{I}, \mathcal{I}'$ with the respective permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ in domain $\mathcal{D}$. The algorithm computes a common permissible partition $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ for the inputs and then refactors $\mathcal{I}, \mathcal{I}'$ with respect to $\overline{\pi}_{\text{common}}$ producing $\mathcal{I}^r, \mathcal{I}'^r$ respectively. The output $\mathcal{I}''$ is computed by applying $T_{\sqcap}$ on individual factors of $\mathcal{I}^r, \mathcal{I}'^r$ separately which reduces its complexity.

The following example illustrates the decomposition of a meet transformer in the Octahedron domain using Algorithm 3.

Example 6.4. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\text{octahedron}} = (\mathbb{U}^4, \{\le, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{\{x_1 \le 1\}, \{x_2 \le 0\}, \{x_3 + x_4 \le 1\}\}, \mathcal{I}' = \{\{x_1 - x_3 - x_4 \le 2\}, \{x_2 \le 1\}\},$$
$$\text{with } \overline{\pi}_{\mathcal{I}} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\} \text{ and } \overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_3, x_4\}, \{x_2\}\}.$$

Algorithm 3 computes a common partition $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_3, x_4\}, \{x_2\}\}$ and refactors $\mathcal{I}, \mathcal{I}'$ with respect to $\overline{\pi}_{\text{common}}$ producing $\mathcal{I}^r, \mathcal{I}'^r$ respectively:

$$\mathcal{I}^r = \{\{x_1 \le 1, x_3 + x_4 \le 1\}, \{x_2 \le 0\}\}, \mathcal{I}'^r = \{\{x_1 - x_3 - x_4 \le 2\}, \{x_2 \le 1\}\}.$$

It then computes the output $\mathcal{I}''$ by applying $T_\sqcap$ on each individual factor $\mathcal{I}^r, \mathcal{I}'^r$ separately:

$$\mathcal{I}'' = \{T_\sqcap(\mathcal{I}^r_1, \mathcal{I}'^r_1), T_\sqcap(\mathcal{I}^r_2, \mathcal{I}'^r_2)\} = \{\{x_1 \le 1, x_3 + x_4 \le 1, x_1 - x_3 - x_4 \le 2\}, \{x_2 \le 0\}\}$$
$$\text{with } \overline{\pi}_{\mathcal{I}''} = \{\{x_1, x_2, x_3\}, \{x_4\}\}.$$

By construction, the decomposed transformer is in $[\![\sqcap, \mathcal{D}]\!]$; it remains to show soundness.

Theorem 6.6. *Let $T_\sqcap$ be a meet transformer. Then the associated decomposed transformer $T^D_\sqcap$ (Algorithm 3) is sound, i.e., $\gamma(T^{best}_\sqcap(\mathcal{I}, \mathcal{I}')) \subseteq \gamma(T^D_\sqcap(\mathcal{I}, \mathcal{I}'))$ for all $\mathcal{I}, \mathcal{I}'$.*

Proof. The effect of applying $T_\sqcap$ on $\mathcal{I}, \mathcal{I}'$ is equivalent to adding each $\iota \in \mathcal{I}'^r$ to $\mathcal{I}^r$ using a conditional transformer T_{cond} in $\mathcal{D}$. Since both $\mathcal{I}^r, \mathcal{I}'^r$ are partitioned according to $\overline{\pi}_{\text{common}}$, there exists a block $\mathcal{B} \in \overline{\pi}_{\text{common}}$ such that $\mathcal{B} \supseteq \mathcal{B}_{\text{cond}}$ for a given ι where $\mathcal{B}_{\text{cond}}$ is the set of variables with non-zero coefficient in ι. Let $\mathcal{I}^r_{\mathcal{B}}$ be the factor corresponding to $\mathcal{B}$ so that $\mathcal{I}^r = \mathcal{I}^r_{\mathcal{B}} \cup \mathcal{I}^r_{\mathcal{U}_c}$ where $\mathcal{I}^r_{\mathcal{U}_c}$ contains the set of constraints not in $\mathcal{I}^r$. For each ι we have,

$$T^{\text{best}}_{\text{cond}}(\mathcal{I}^r, \iota) = T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\mathcal{B}} \cup \mathcal{I}^r_{\mathcal{U}_c}) = (\mathcal{I}^r_{\mathcal{B}} \cup \iota) \cup \mathcal{I}^r_{\mathcal{U}_c} = T^{\text{best}}_{\text{cond}}(\mathcal{I}^r_{\mathcal{B}}) \cup \mathcal{I}^r_{\mathcal{U}_c} \sqsubseteq T_{\text{cond}}(\mathcal{I}^r_{\mathcal{B}}) \cup \mathcal{I}^r_{\mathcal{U}_c}.$$

Thus, the theorem holds. □

$T_\sqcap$ in Algorithm 3 does not create constraints between the variables in different blocks of the common partition in the output $\mathcal{I}''$. From Theorem 6.6, it follows that $\overline{\pi}_{\mathcal{I}''} = \overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$. Since the exact syntactic form of $\mathcal{I}''$ is not known, $\overline{\pi}_{\mathcal{I}''} \neq \pi_{\mathcal{I}''}$. The following corollary provides conditions to check when the output partition $\overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''}$.

Corollary 6.7. *$\overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''}$ if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ and $\mathcal{I}'' = \mathcal{I} \cup \mathcal{I}'$.*

6.4 Join (⊔)

Algorithm 4 shows our construction for a join transformer $T_\sqcup \in [\![\sqcup, \mathcal{D}]\!]$ on input elements $\mathcal{I}, \mathcal{I}'$ with the respective permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ in domain $\mathcal{D}$. The algorithm computes a common permissible partition $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ and refactors $\mathcal{I}, \mathcal{I}'$ with respect to this partition producing $\mathcal{I}^r$ and $\mathcal{I}'^r$ respectively. For each pair of factors $\mathcal{I}^r_k, \mathcal{I}'^r_k$, the algorithm checks whether they are equal. If the equality holds, then the algorithm adds $\mathcal{I}^r_k$ to the output $\mathcal{I}''$ and adds the corresponding block $\mathcal{X}_k$ to the partition $\overline{\pi}$. The algorithm combines the factors which are not equal by taking union into bigger factors $\mathcal{I}^r_\sqcup, \mathcal{I}'^r_\sqcup$ respectively. It combines the corresponding blocks by taking union to form the set $\mathcal{N}$. The algorithm then applies $T_\sqcup$ on factors $\mathcal{I}^r_\sqcup, \mathcal{I}'^r_\sqcup$ which reduces its complexity. Finally, the algorithm adds $\mathcal{N}$ to $\overline{\pi}$.

The following example illustrates the decomposition of a join transformer in the Octagon domain using Algorithm 4.

Algorithm 4 Join transformer

function JOIN($\mathcal{I}, \mathcal{I}', \overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}, T_{\sqcup}$)
  Parameters:
    $\mathcal{I} \leftarrow \{\mathcal{I}_1, \mathcal{I}_2, \dots, \mathcal{I}_p\}$
    $\mathcal{I}' \leftarrow \{\mathcal{I}'_1, \mathcal{I}'_2, \dots, \mathcal{I}'_q\}$
    $\overline{\pi}_{\mathcal{I}} \leftarrow \{\mathcal{X}_{\mathcal{I}_1}, \mathcal{X}_{\mathcal{I}_2}, \dots, \mathcal{X}_{\mathcal{I}_p}\}$
    $\overline{\pi}_{\mathcal{I}'} \leftarrow \{\mathcal{X}_{\mathcal{I}'_1}, \mathcal{X}_{\mathcal{I}'_2}, \dots, \mathcal{X}_{\mathcal{I}'_q}\}$
    $T_{\sqcup} \leftarrow$ join transformer
  $\overline{\pi}_{\text{common}} := \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'} := \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_l\}$
  $\mathcal{I}^r :=$ refactor($\mathcal{I}, \overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\text{common}}$)
  $\mathcal{I}'^r :=$ refactor($\mathcal{I}', \overline{\pi}_{\mathcal{I}'}, \overline{\pi}_{\text{common}}$)
  $\mathcal{I}'' := \emptyset$
  $\overline{\pi}_{\mathcal{I}''} = \emptyset$
  $\mathcal{N} = \emptyset$
  $\mathcal{I}_{\sqcup} := \mathcal{I}'_{\sqcup} := \emptyset$
  for $k \in \{1, 2, \dots, l\}$ **do**
    if $\mathcal{I}^r_k = \mathcal{I}'^r_k$ **then**
      $\mathcal{I}''$.add($\mathcal{I}^r_k$)
      $\overline{\pi}_{\mathcal{I}''}$.add($\mathcal{X}_k$)
    else
      $\mathcal{I}_{\sqcup} := \mathcal{I}_{\sqcup} \cup \mathcal{I}^r_k$
      $\mathcal{I}'_{\sqcup} := \mathcal{I}'_{\sqcup} \cup \mathcal{I}'^r_k$
      $\mathcal{N} := \mathcal{N} \cup \mathcal{X}_k$
  $\mathcal{I}''$.add($T_{\sqcup}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup})$)
  $\overline{\pi}_{\mathcal{I}''}$.add($\mathcal{N}$)

Example 6.5. Consider

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X},\text{Octagon}} : (\mathbb{U}^2 \times \{0\}, \{\leq, =\}, \mathbb{R}), \\ \mathcal{I} &= \{\{x_1 \leq 2\}, \{x_2 \leq 1\}, \{x_3 \leq 3\}\}, \mathcal{I}' = \{\{x_1 \leq 1\}, \{x_2 \leq 3\}, \{x_3 \leq 3\}\} \text{ with} \\ \overline{\pi}_{\mathcal{I}} &= \overline{\pi}_{\mathcal{I}'} = \{\{x_1\}, \{x_2\}, \{x_3\}\}. \end{aligned}$$

Since $\overline{\pi}_{\mathcal{I}} = \overline{\pi}_{\mathcal{I}'}$, Algorithm 4 does not refactor $\mathcal{I}$ and $\mathcal{I}'$. Here we have, $\mathcal{I}_1 \neq \mathcal{I}'_1$, $\mathcal{I}_2 \neq \mathcal{I}'_2$ and $\mathcal{I}_3 = \mathcal{I}'_3$. Thus the algorithm combines $\mathcal{I}_1, \mathcal{I}_2$ into a single factor $\mathcal{I}_{\sqcup}$. Similarly, it combines $\mathcal{I}'_1, \mathcal{I}'_2$ into $\mathcal{I}'_{\sqcup}$:

$$\mathcal{I}_{\sqcup} = \{x_1 \leq 2, x_2 \leq 1\}, \mathcal{I}'_{\sqcup} = \{x_1 \leq 1, x_2 \leq 3\}.$$

The algorithm applies $T^{\text{best}}_{\sqcup}$ only on $\mathcal{I}_{\sqcup}$ and $\mathcal{I}'_{\sqcup}$ whereas $\mathcal{I}_3$ is added to the output directly:

$$\mathcal{I} \sqcup \mathcal{I}' = \{\{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4\}, \{x_3 \leq 3\}\} \text{ with } \overline{\pi}_{\mathcal{I} \sqcup \mathcal{I}'} = \{\{x_1, x_2\}, \{x_3\}\}.$$

By construction the decomposed transformer is in $[\![\sqcup, \mathcal{D}]\!]$; it remains to show soundness.

THEOREM 6.8. *Let $T_{\sqcup} \in [\![\sqcup, \mathcal{D}]\!]$ be a join transformer. Then the associated decomposed transformer $T^D_{\sqcup}$ (Algorithm 4) is sound, i.e., $\gamma(T^{best}_{\sqcup}(\mathcal{I}, \mathcal{I}')) \subseteq \gamma(T^D_{\sqcup}(\mathcal{I}, \mathcal{I}'))$ for all $\mathcal{I}, \mathcal{I}'$.*

PROOF. By construction $\mathcal{I}^r = \mathcal{I}$ and $\mathcal{I}'^r = \mathcal{I}'$. Algorithm 4 applies $T_{\sqcup}$ on the factors $\mathcal{I}_{\sqcup}$ and $\mathcal{I}'_{\sqcup}$ corresponding to the block $\mathcal{N}$. Let $\mathcal{M} = \mathcal{X} \setminus \mathcal{N}$ and $\mathcal{I}_{\mathcal{M}}, \mathcal{I}'_{\mathcal{M}}$ be the corresponding factors. We can write $T^D_{\sqcup}(\mathcal{I}^r, \mathcal{I}'^r) = T_{\sqcup}(\beta_{\mathcal{N}}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup})) \cup \mathcal{I}_{\mathcal{M}}$. We know that $\mathcal{I}_{\mathcal{M}} = \mathcal{I}'_{\mathcal{M}}$ and thus $T^{\text{best}}_{\sqcup}(\mathcal{I}_{\mathcal{M}}, \mathcal{I}'_{\mathcal{M}}) = \mathcal{I}_{\mathcal{M}}$.

$$\begin{aligned} T^{\text{best}}_{\sqcup}(\mathcal{I}^r, \mathcal{I}'^r) &= T^{\text{best}}_{\sqcup}(\mathcal{I}_{\sqcup} \cup \mathcal{I}_{\mathcal{M}}, \mathcal{I}'_{\sqcup} \cup \mathcal{I}'_{\mathcal{M}}) \\ &\sqsubseteq \beta_{\mathcal{N}}(T^{\text{best}}_{\sqcup}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup})) \times \beta_{\mathcal{M}}(T^{\text{best}}_{\sqcup}(\mathcal{I}_{\mathcal{M}}, \mathcal{I}'_{\mathcal{M}})) \\ &= \beta_{\mathcal{N}}(T^{\text{best}}_{\sqcup}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup})) \times \beta_{\mathcal{M}}(\mathcal{I}_{\mathcal{M}}) \\ &= T^{\text{best}}_{\sqcup}(\beta_{\mathcal{N}}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup}))) \cup \mathcal{I}_{\mathcal{M}} \\ &\sqsubseteq T_{\sqcup}(\beta_{\mathcal{N}}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup})) \cup \mathcal{I}_{\mathcal{M}} \\ &= T^D_{\sqcup}(\mathcal{I}^r, \mathcal{I}'^r). \end{aligned}$$

Since γ is monotone, we have $\gamma(T^{\text{best}}_{\text{cond}}(\mathcal{I})) \subseteq \gamma(T^D_{\text{cond}}(\mathcal{I}))$ and thus the theorem holds. □

Algorithm 5 Widening transformer

function WIDENING($\mathcal{I}, \mathcal{I}', \overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}, T_{\triangledown}$)
  Parameters:
    $\mathcal{I} \leftarrow \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_p\}$
    $\mathcal{I}' \leftarrow \{\mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_q\}$
    $\overline{\pi}_{\mathcal{I}} \leftarrow \{\mathcal{X}_{\mathcal{I}_1}, \mathcal{X}_{\mathcal{I}_2}, \ldots, \mathcal{X}_{\mathcal{I}_p}\}$
    $\overline{\pi}_{\mathcal{I}'} \leftarrow \{\mathcal{X}_{\mathcal{I}'_1}, \mathcal{X}_{\mathcal{I}'_2}, \ldots, \mathcal{X}_{\mathcal{I}'_q}\}$
    $T_{\triangledown} \leftarrow$ widening transformer
  $\mathcal{I}'' := \emptyset$
  $\overline{\pi}_{\text{common}} := \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$
  $\mathcal{I}^r :=$ refactor$(\mathcal{I}, \overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\text{common}})$
  $\mathcal{I}'^r :=$ refactor$(\mathcal{I}', \overline{\pi}_{\mathcal{I}'}, \overline{\pi}_{\text{common}})$
  for $k \in \{1, 2, \ldots, l\}$ **do**
    $\mathcal{I}''$.add$(T_{\triangledown}(\mathcal{I}^r_k, \mathcal{I}'^r_k))$
  $\overline{\pi}_{\mathcal{I}''} := \overline{\pi}_{\text{common}}$

As for the conditional and the assignment, an arbitrary $\mathcal{N}$ does not ensure soundness. Algorithm 4 applies $T_{\sqcup}$ only on the factors $\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup}$ corresponding to the block $\mathcal{N}$. Thus the factors corresponding to the other blocks in $\overline{\pi}$ remain unchanged. From this, it follows that $\overline{\pi}_{\mathcal{I}''}$ as computed in Algorithm 4 is permissible for $\mathcal{I}''$.

Refinement. We can refine the output partition $\overline{\pi}_{\mathcal{I}''}$ after computing the output $\mathcal{I}''$ without inspecting $\mathcal{I}''$. For this we need to check the inputs $\mathcal{I}, \mathcal{I}'$. If a variable x_i is unconstrained in either $\mathcal{I}$ or $\mathcal{I}'$, then it is also unconstrained in $\mathcal{I}''$. $\overline{\pi}_{\mathcal{I}''}$ can be refined by removing x_i from the block containing it and adding the singleton set $\{x_i\}$ to $\overline{\pi}_{\mathcal{I}''}$. This refinement can only be performed after applying $T_{\sqcup}$. The following theorem formalizes this refinement:

THEOREM 6.9. *Let $\mathcal{I}, \mathcal{I}'$ be abstract elements in $\mathcal{D}$ with the associated permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ respectively. Let $\mathcal{U} = \{x_i \mid x_i$ is unconstrained in either $\mathcal{I}$ or $\mathcal{I}'\}$. Then the following partition is permissible for the output $\mathcal{I}''$:*

$$\overline{\pi}'_{\mathcal{I}''} = \{\mathcal{N}, \mathcal{X}_1, \ldots, \mathcal{X}_r\} \sqcap \{\mathcal{X} \setminus \mathcal{U}, \{u_1\}, \ldots, \{u_{r'}\}\},$$

where $\mathcal{X}_i \in \mathcal{E}$ and $u_i \in \mathcal{U}$.

The proof of Theorem 6.9 is immediate from the discussion above. Unlike other transformers, we do not know of any conditions for checking whether $\overline{\pi}'_{\mathcal{I}''} = \pi_{\mathcal{I}''}$.

6.5 Widening ($\triangledown$)

Algorithm 5 shows our construction for a widening transformer $T_{\triangledown} \in [\![\triangledown, \mathcal{D}]\!]$ on input elements $\mathcal{I}, \mathcal{I}'$ with the respective permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ in $\mathcal{D}$. The algorithm computes a common permissible partition $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ and refactors $\mathcal{I}, \mathcal{I}'$ with respect to this partition producing $\mathcal{I}^r$ and $\mathcal{I}'^r$ respectively. The widening transformer $T_{\triangledown}$ is then applied on each factor $\mathcal{I}^r_k, \mathcal{I}'^r_k$ separately which reduces its complexity.

The following example illustrates the decomposition of the standard semantic TVPI widening transformer using Algorithm 5.

Example 6.6. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\text{tvpi}} = (\mathbb{Z}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}),$$

$$\mathcal{I} = \{\{x_1 \leq 1\}, \{x_2 \leq 0\}, \{x_3 + x_4 \leq 1\}\}, \mathcal{I}' = \{\{2x_1 - 3x_2 \leq 2, x_1 + x_2 \leq 1\}, \{x_3 \leq 0\}, \{x_4 \leq 0\}\},$$

with $\overline{\pi}_{\mathcal{I}} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\}$ and $\overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\}$.

Algorithm 5 computes a common permissible partition $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_2\}, \{x_3, x_4\}\}$ and then refactors $\mathcal{I}, \mathcal{I}'$ with respect to $\overline{\pi}_{\text{common}}$ yielding $\mathcal{I}^r, \mathcal{I}'^r$ respectively:

$$\mathcal{I}^r = \{\{x_1 \leq 1, x_2 \leq 0\}, \{x_3 + x_4 \leq 1\}\}, \mathcal{I}'^r = \{\{2x_1 - 3x_2 \leq 2, x_1 + x_2 \leq 1\}, \{x_3 \leq 0, x_4 \leq 0\}\}.$$

It then computes the output $\mathcal{I}''$ by applying $T_\triangledown$ on individual factors $\mathcal{I}^r, \mathcal{I}''^r$ separately: $\mathcal{I}'' = \{T_\triangledown(\mathcal{I}_1^r, \mathcal{I}_1''^r), T_\triangledown(\mathcal{I}_2^r, \mathcal{I}_2''^r)\} = \{\{x_1 + x_2 \le 1\}, \{x_3 + x_4 \le 1\}\}$ with $\overline{\pi}_{\mathcal{I}''} = \{\{x_1, x_2\}, \{x_3, x_4\}\}$.

In contrast to prior transformers, for a general widening transformer the construction of the decomposed transformer is not sound in general. Thus we have to require that the given widening transformer is already in $[\![\triangledown, \mathcal{D}]\!]$, which trivially guarantees soundness. Standard widening transformers satisfy this condition.

$T_\triangledown$ in Algorithm 5 does not create constraints between variables in different blocks of the common partition in the output $\mathcal{I}''$. By construction, it follows that $\overline{\pi}_{\mathcal{I}''} = \overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$. For syntactic widening, the output partition $\overline{\pi}_{\mathcal{I}''}$ can be refined to $\overline{\pi}_{\mathcal{I}}$ after computing the output $\mathcal{I}''$. The following corollaries provide conditions when $\overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''}$ for the semantic and the syntactic widening respectively.

Corollary 6.10. *For semantic widening,* $\overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''}$ *if* $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ *and* $\mathcal{I}'' = \mathcal{I} \cup \mathcal{I}'$.

Corollary 6.11. *For syntactic widening,* $\overline{\pi}_{\mathcal{I}''} = \pi_{\mathcal{I}''}$ *if* $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$ *and* $\mathcal{I}'' = \mathcal{I}$.

7 EXPERIMENTAL EVALUATION

In this section we evaluate the performance of our generic decomposition approach on three popular domains: Polyhedra, Octagons, and Zones. Using standard implementations of these domains, we show that our decomposition of their transformers leads to substantial performance improvements, often surpassing existing transformers designed for specific domains.

Experimental Setup. All of our experiments were performed on a 3.5 GHz Intel Quad Core i7-4771 Haswell CPU. The machine has L1, L2, and L3 caches of sizes 256 KB, 1024 KB, and 8192 KB, respectively, while main memory has 16 GB. Turbo boost and Hyper threading were disabled for consistency of measurements. All libraries were compiled with gcc 5.2.1 using the flags `-O3 -m64 -march=native`. We used a time limit of 4 hours for our experiments.

Benchmarks. The benchmarks for our experiments were taken from the popular software verification competition (Beyer 2016). The benchmark suite is divided into categories suited for different kinds of analysis e.g., pointer, array, numerical etc. We chose two categories suited for numerical analysis: (i) Linux Device Drivers (LD), and (ii) Control Flow (CF). Each of these categories contains hundreds of benchmarks and we evaluated the performance of our analysis on each of these. We use the crab-llvm analyzer which is part of the SeaHorn verification framework (Gurfinkel et al. 2015) for performing the analysis. The analyzer is written in C++ and performs intraprocedural analysis of LLVM bitcode. The analyzer explicitly checks for unconstrained variables during runtime and removes them. Thus, the total number of variables for Polyhedra, Octagon, and Zones can be different on the same benchmark.

7.1 Polyhedra

The standard implementation of the Polyhedra domain is based on the double representation method, i.e., it keeps both the constraints and the generator representation. This is because transformers such as meet are cheap with the constraint representation but expensive with the generator representation. On the other hand transformers such as join are cheap with the generator representation but expensive with the constraint representation. The Polyhedra analysis thus applies the domain transformer on one representation and then updates the other representation using a standard conversion algorithm (Chernikoba 1968; Verge 1994). The standard implementation contains the best conditional, assignment, meet and join transformers together wtih a (semantic) widening operator. All of these transformers satisfy the (decomposable) definitions from Section 5.

Table 2. Asymptotic time complexity of the Polyhedra transformers with and without decomposition.

Transformer	Non-Decomposed	Decomposed
Conditional	$O(n)$	$O(n_{\max})$
Assignment	$O(ng)$	$O(n_{\max}g_{\max})$
Meet ($\sqcap$)	$O(nm)$	$O(\sum_{i=1}^{l} n_i m_i)$
Join ($\sqcup$)	$O(ng)$	$O(\sum_{i=1}^{l} n_i g_i m_i + n_{\max}g_{\max})$
Widening (∇)	$O(ngm)$	$O(\sum_{i=1}^{l} n_i g_i m_i)$
Conversion	$O(exp(n,g))$	$O(\sum_{i=1}^{l} exp(n_i, g_i))$

Table 2 shows the asymptotic complexity of Polyhedra transformers in the standard implementation with and without decomposition (Singh et al. 2017). For the non-decomposed column in the table, n is the number of variables, m is the number of constraints and g is the number of generators whereas for the decomposed column, l is the number of blocks in the partition, n_i is the number of variables in the i-th block, $n_{\max}$ is the number of variables in the largest block, m_i and g_i are the number of constraints and generators respectively in the i-th factor and g_{max} is the number of generators in the largest factor. It holds that $n = \sum_{i=1}^{l} n_i$, $m = \sum_{i=1}^{l} m_i$ and $g = \prod_{i=1}^{n} g_i$. We also show the complexity of the conversion algorithm for converting from the constraints to the generators. It has the same exponential complexity (in terms of n and g) for the conversion in the other direction. Thus, it is the most expensive operation in the standard implementation.

We compare the runtime and memory consumption for the end-to-end Polyhedra analysis with our generic decomposed transformers versus the original non-decomposed transformers from Parma Polyhedra Library (PPL) (Bagnara et al. 2008) and the decomposed transformers from ELINA (Singh et al. 2017). PPL, ELINA and our decomposition store the constraints and the generators using matrices with 64-bit integers. PPL stores a single matrix for either representation whereas both ELINA and our decomposition use a set of matrices corresponding to the factors. It can require exponential space in the worst case to store the representations. Table 3 shows the results on 13 large, representative benchmarks. These benchmarks were chosen based on the following criteria:

- The most time consuming function in the benchmark did not produce any integer overflow with PPL, ELINA, or our approach.
- The benchmark ran for at least 2 minutes with PPL.

Our decomposition maintains semantic equivalence with both ELINA and PPL as long as there is no integer overflow. All three implementations set the polyhedron to $\top$ whenever an integer overflow occurs. The total number of integer overflows on the chosen benchmarks were 58, 23 and 21 for PPL, ELINA, and our decomposition, respectively. We also had fewer integer overflows than both ELINA and PPL on the remaining benchmarks. Thus, our decomposition improves in some cases also the precision of the analysis with respect to both ELINA and PPL.

Table 3 shows our experimental findings. The entry *MO* (memory-out) in the table means that the analysis ran out of memory whereas the entry *TO* (time-out) means the analysis did not finish within 4 hours. Whenever there is memory overflow and our analysis finishes, we show the corresponding speedup as ∞, because the analysis can never finish on the given machine even if given arbitrary time. We specify lower bounds for the speedups in case of a time-out.

In the table, PPL either ran out of memory or did not finish within 4 hours on 8 out of the 13 benchmarks. Both ELINA and our decomposition are able to analyze all benchmarks. We are faster than ELINA on all benchmarks. We achieve speedup over ELINA on all benchmarks with the maximum speedup being 5.9x on the P19_159 benchmark. It can also be seen that our

Table 3. Speedup for the Polyhedra domain analysis with our decomposition over PPL and ELINA.

Benchmark	PPL		ELINA		Our Decomposition		Speedup vs.	
	time(s)	memory(GB)	time(s)	memory(GB)	time(s)	memory(GB)	PPL	ELINA
`firewire_firedtv`	331	0.9	0.4	0.2	0.2	0.2	1527	2
`net_fddi_skfp`	6142	7.2	9.2	0.9	4.4	0.3	1386	2
`mtd_ubi`	MO	MO	4	0.9	1.9	0.3	∞	2.1
`usb_core_main0`	4003	1.4	65	2	29	0.7	136	2.2
`tty_synclinkmp`	MO	MO	3.4	0.1	2.5	0.1	∞	1.4
`scsi_advansys`	TO	TO	4	0.4	3.4	0.2	>4183	1.2
`staging_vt6656`	TO	TO	2	0.4	0.5	0.1	>28800	4
`net_ppp`	10530	0.1	924	0.3	891	0.1	11.8	1
`p10_l00`	121	0.9	11	0.8	5.4	0.2	22.4	2
`p16_l40`	MO	MO	11	3	2.9	0.4	∞	3.8
`p12_l57`	MO	MO	14	0.8	6.5	0.3	∞	2.1
`p13_l53`	MO	MO	54	2.7	25	0.9	∞	2.2
`p19_l59`	MO	MO	70	1.7	12	0.6	∞	5.9

decomposition saves significant memory over ELINA. The speedups on the remaining benchmarks over the decomposed version of ELINA varies from 1.1x up to 4x.

Table 4. Partition statistics for the Polyhedra domain analysis.

Benchmark	Category	LOC	n		$n_{\text{max}}^{\text{elina}}$		$n_{\text{max}}^{\text{our}}$		$n_{\text{max}}^{\text{finest}}$	
			max	avg	max	avg	max	avg	max	avg
`firewire_firedtv`	LD	14506	159	25	81	7	40	4	39	3
`net_fddi_skfp`	LD	30186	589	88	111	25	45	9	13	4
`mtd_ubi`	LD	39334	528	59	111	14	28	5	23	4
`usb_core_main0`	LD	52152	365	72	267	30	60	11	40	7
`tty_synclinkmp`	LD	19288	332	49	48	10	40	6	26	4
`scsi_advansys`	LD	21538	282	63	117	18	49	12	41	9
`staging_vt6656`	LD	25340	675	53	204	17	25	4	12	3
`net_ppp`	LD	15744	218	58	112	40	51	28	43	20
`p10_l00`	CF	592	303	174	234	54	79	16	14	6
`p16_l40`	CF	1783	874	266	86	31	39	14	5	3
`p12_l57`	CF	4828	921	261	461	78	21	7	4	3
`p13_l53`	CF	5816	1631	342	617	111	26	10	9	3
`p19_l59`	CF	9794	1272	358	867	187	31	8	12	3

Better partitioning leads to performance improvements. Table 4 shows further statistics for the category (LD or CF) and the number of lines of code in each benchmark. As can be seen, the benchmarks are quite large and contain up to 50K lines of code. After each join, we measured the total number of variables (which is the same for all benchmarks) n and report the maximum and the average. For the decomposed analyses (ELINA and ours) we measured the size of the largest block and report again maximum and average under $n_{\text{max}}^{\text{elina}}$, $n_{\text{max}}^{\text{our}}$. To assess the quality of the partitions, we also computed (with the needed overhead) the finest partition after each join and show the largest blocks under $n_{\text{max}}^{\text{finest}}$ (maximum and average). As can be observed, our partitions are strictly finer than the ones produced by ELINA on all benchmarks due to the refinements for the assignment and the join transformer. Moreover, it can be seen that our partitions are sometimes close to the finest partition but in many cases there is room for further improvement.

7.2 Octagon

The standard implementation of the Octagon domain approximates the best conditional and best assignment transformers whereas it implements the best join and meet transformers. The widening is defined syntactically. All of these transformers satisfy the definitions in Section 5. The implementation keeps only the constraint representation. The implementation also requires strong closure operation for the efficiency and precision of transformers such as join, conditional, assignment, etc.

Table 5. Asymptotic time complexity of the Octagon transformers with and without decomposition.

Transformer	Non-Decomposed	Decomposed
Conditional	$O(n^2)$	$O(n_{\max}^2)$
Assignment	$O(n^2)$	$O(n_{\max}^2)$
Meet (⊓)	$O(n^2)$	$O(\sum_{i=1}^{l} n_i^2)$
Join (⊔)	$O(n^2)$	$O(\sum_{i=1}^{l} n_i^2)$
Widening (∇)	$O(n^2)$	$O(\sum_{i=1}^{l} n_i^2)$
Strong Closure	$O(n^3)$	$O(\sum_{i=1}^{l} n_i^3)$

Table 5 shows the asymptotic complexity of standard Octagon transformers as well as the strong closure operation with and without decomposition (Singh et al. 2015). In the table $n, n_i, n_{\max}$ have the same meaning as in Table 2. In can be seen that strong closure is the most expensive operation in this domain (it has cubic complexity). It is possible to apply it incrementally for the conditional and the assignment transformers.

We compare the performance of our approach against the standard Octagon analysis, using the non-decomposed ELINA (ELINA-ND) and the decomposed (ELINA-D) transformers from ELINA. All of these implementations store the constraint representation using a single matrix with 64-bit doubles. The matrix requires quadratic space in terms of n. Thus, overall memory consumption is the same for all implementations. We compare the runtime and report speedups for the end-to-end Octagon analysis in Table 6. We achieve up to 40x speedup for the end-to-end analysis over the non-decomposed implementation. More importantly, we are faster than the decomposed version of ELINA on all benchmarks bar one. The maximum speedup over the decomposed version of ELINA is 2.2x. The speedups on the remaining benchmarks vary between 1x to 1.6x.

Table 7 shows the partition statistics for the Octagon analysis (as we did for the Polyhedra analysis). It can be seen that while our refinements often produce finer partitions than the decomposed version of ELINA, they are coarser on 3 of the 13 benchmarks. This is because the decomposed transformers in ELINA are quite specialized for the standard approximations of the conditional and assignment transformers. We still achieve comparable performance on these benchmarks. Note that the partitions are quite close to the finest in most cases.

7.3 Zones

The standard conditional and assignment transformers in the Zones domain are approximate whereas the meet and join are the best transformer (Miné 2002). The widening is defined syntactically. All of these transformers satisfy the definitions in Section 5. The transformers require only the constraint representation. As for the Octagon domain, a cubic closure operation is required. The domain transformers have the same asymptotic complexity as the Octagon domain.

Table 6. Speedup for the Octagon domain analysis with our decomposition over the non-decomposed and the decomposed versions of ELINA.

Benchmark	ELINA-ND	ELINA-D	Our Decomposition	Speedup vs.	
	time(s)	time(s)	time(s)	ELINA-ND	ELINA-D
firewire_firedtv	0.4	0.07	0.07	5.7	1
net_fddi_skfp	28	2.6	1.9	15	1.4
mtd_ubi	3411	979	532	6.4	1.8
usb_core_main0	107	6.1	4.9	22	1.2
tty_synclinkmp	8.2	1	0.8	10	1.2
scsi_advansys	9.3	1.5	0.8	12	1.9
staging_vt6656	4.8	0.3	0.2	24	1.5
net_ppp	11	1.1	1.2	9.2	0.9
p10_l00	20	0.5	0.5	40	1
p16_l40	8.8	0.6	0.5	18	1.2
p12_l57	19	1.2	0.7	27	1.7
p13_l53	43	1.7	1.3	33	1.3
p19_l59	41	2.8	1.2	31	2.2

Table 7. Partition statistics for the Octagon domain analysis.

Benchmark	Category	LOC	n		n_{max}^{elina}		n_{max}^{our}		n_{max}^{finest}	
			max	avg	max	avg	max	avg	max	avg
firewire_firedtv	LD	14506	159	25	31	6	40	4	27	3
net_fddi_skfp	LD	30186	573	86	49	18	30	10	14	7
mtd_ubi	LD	39334	553	46	111	65	22	9	16	9
usb_core_main0	LD	52152	364	72	59	22	39	9	35	7
tty_synclinkmp	LD	19288	324	49	84	15	26	6	25	4
scsi_advansys	LD	21538	293	64	94	19	41	6	20	5
staging_vt6656	LD	25340	651	52	63	7	25	4	14	3
net_ppp	LD	15744	218	54	40	23	55	29	39	19
p10_l00	CF	592	305	173	19	10	77	16	17	9
p16_l40	CF	1783	874	266	32	12	13	7	10	5
p12_l57	CF	4828	954	265	55	15	13	4	11	4
p13_l53	CF	5816	1635	337	41	12	22	7	10	5
p19_l59	CF	9794	1291	363	79	14	22	4	18	3

We implemented both, a non-decomposed version of the transformers as well as a version with our decomposition method of the standard transformers. Both implementations store the constraints using a single matrix with 64-bit doubles that requires quadratic space. We compare the runtime and report speedups for the end-to-end Zones analysis in Table 8. Our decomposition achieves speedup up to 6x over the non-decomposed implementation. The speedups over the remaining benchmarks not shown in the table also vary from 1.1x up to 5x.

Table 9 shows the partition statistics for the Zones analysis. It can be seen that partitioning works (and is the core reason for the speed-ups) and the obtained partitions are close to the finest.

Summary. Overall, we can see that the generic decomposition proposed in this paper is a suitable construction for speeding-up analysis with numerical domains. We also show that the partitions

Table 8. Speedup for the Zones domain analysis with our decomposition over non-decomposed implementation.

Benchmark	Non-Decomposed time(s)	Our Decomposition time(s)	Speedup vs. Non-Decomposed
firewire_firedtv	0.05	0.05	1
net_fddi_skfp	3	1.5	2
mtd_ubi	1.4	0.7	2
usb_core_main0	10.3	4.6	2.2
tty_synclinkmp	1.1	0.7	1.6
scsi_advansys	0.9	0.7	1.3
staging_vt6656	0.5	0.2	2.5
net_ppp	1.1	0.7	1.5
p10_l00	1.9	0.4	4.6
p16_l40	1.7	0.7	2.5
p12_l57	3.5	0.9	3.9
p13_l53	8.7	2.1	4.2
p19_l59	9.8	1.6	6.1

Table 9. Partition statistics for the Zones domain analysis.

Benchmark	Category	LOC	n		$n_{\max}^{\text{our}}$		$n_{\max}^{\text{finest}}$	
			max	avg	max	avg	max	avg
firewire_firedtv	LD	14506	159	25	40	4	17	3
net_fddi_skfp	LD	30186	578	88	30	9	13	5
mtd_ubi	LD	39334	553	59	23	5	14	3
usb_core_main0	LD	52152	362	71	37	8	33	7
tty_synclinkmp	LD	19288	328	49	26	6	25	5
scsi_advansys	LD	21538	293	65	41	8	21	7
staging_vt6656	LD	25340	675	53	25	3	13	2
net_ppp	LD	15744	219	58	54	29	47	24
p10_l00	CF	592	303	174	77	16	17	8
p16_l40	CF	1783	856	261	13	7	10	6
p12_l57	CF	4828	882	249	12	4	10	3
p13_l53	CF	5816	1557	317	22	7	20	5
p19_l59	CF	9794	1243	331	14	4	13	3

computed during analysis are close to optimal for Octagons and Zones but with further room for improvement for Polyhedra. The challenge is how to obtain those with reasonable cost. Further speed-ups can also be obtained by different implementations of the transformers that are, for example, selectively approximate to further partition.

8 RELATED WORK

We have discussed dynamic partitioning specialized for the standard implementations of the subpolyhedra domains throughout the paper (Halbwachs et al. 2003; Singh et al. 2015, 2017). In this section, we discuss other related work which is related to increasing the performance of numerical domain analysis.

Variable packing (Blanchet et al. 2003; Heo et al. 2016) has been used for decomposing the Octagon transformers. $\mathcal{X}$ is partitioned statically before running the analysis based on a critieria, e.g., if two variables are in the same pack if they are in the same program statement. Although the variable packing can be generalized to decompose the transformers of other domains, more precise results for the transformers can be obtained by maintaining partitions dynamically. For example, the join transformer usually relates the variables that do not occur in the same program statement and thus variable packing is bound to lose precision. The work of (Venet and Brat 2004) dynamically maintains partitions based on a syntactic criteria for the Zones domain. The generated transformers are less precise than the ones generated using our approach.

The work of (Gange et al. 2016) and (Jourdan 2017) is focussed on designing sparse algorithms for the standard transformers of the Zones and the Octagon domain respectively. While these algorithms cannot be extended to more expressive domains, they can certainly be combined with our decomposition to potentially achieve better performance.

Both (Simon and King 2005) and (Miné et al. 2010) focus on improving the performance of the best join transformer in the Polyhedra domain based on the constraint representation. In (Simon and King 2005) the authors exploit sparsity by noticing that a given variable occurs only a few times in the constraint representations of the Polyhedra. If the output becomes too large, they approximate. Frequent calls to the linear solve limit the performance of their approach. In (Miné et al. 2010) the authors decompose the best join transformer by decomposing the inputs into two pieces each. The join transformer is then applied on one of the pieces. The partitions obtained with this method are very coarse and thus the decomposed transformer has worse performance than achieved using our decomposition.

9 CONCLUSION

Partitioning abstract elements is a promising avenue to make abstract domain analysis faster, possibly by orders of magnitude, and thus practical for many real world verification tasks. It is possible due to the inherent "locality" in the way program statements, and sequences of such, access variables. This paper advances partitioning by showing that it is applicable to all sub-polyhedra domains and shows how to construct decomposed transformers from existing, non-decomposed transformers. This way, existing implementations can be re-factored to incorporate decomposition. The construction provides guarantees on the quality of the achievable partitions. Finally, we provide techniques to refine the partitions of the output of important transformers in certain cases, which improves over prior work. We evaluated our approach on three expensive abstract domains: Zones, Octagons, and Polyhedra and show significant speed-ups compared to prior work, including domains that were previously decomposed manually.

REFERENCES

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21.

Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 887–904.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*. 196–207.

N.V. Chernikoba. 1968. Algorithm for discovering the set of all the solutions of a linear programming problem. *U. S. S. R. Comput. Math. and Math. Phys.* 8, 6 (1968), 282 – 293.

Robert Claris and Jordi Cortadella. 2007. The octahedron abstract domain. *Science of Computer Programming* 64 (2007), 115 – 139.

Patrick Cousot and Radhia Cousot. 1976. *Static determination of dynamic properties of programs.* 106–130.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. Symposium on Principles of Programming Languages (POPL)*. 84–96.

Radhia Cousot, Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. 2005. Precise widening operators for convex polyhedra. *Science of Computer Programming* 58, 1 (2005), 28 – 56.

Pietro Ferrara, Francesco Logozzo, and Manuel Fähndrich. 2008. Safer Unsafe Code for .NET. *SIGPLAN Not.* 43 (2008), 329–346.

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. *Exploiting Sparsity in Difference-Bound Matrices.* 189–211.

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Proc. Computer Aided Verification (CAV)*. 343–361.

N. Halbwachs, D. Merchat, and L. Gonnord. 2006. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design (FMSD)* 29, 1 (2006), 79–95.

Nicolas Halbwachs, David Merchat, and Catherine Parent-Vigouroux. 2003. *Cartesian Factoring of Polyhedra in Linear Relation Analysis.* 355–365.

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *Proc. Static Analysis Symposium (SAS)*. 237–256.

Jacob M. Howe and Andy King. 2009. *Logahedra: A New Weakly Relational Domain.* 306–320.

Jacques-Henri Jourdan. 2017. Sparsity Preserving Algorithms for Octagons. *Electronic Notes in Theoretical Computer Science* 331 (2017), 57 – 70. Workshop on Numerical and Symbolic Abstract Domains (NSAD).

Michael Karr. 1976. Affine relationships among variables of a program. *Acta Informatica* 6 (1976), 133–151.

Francesco Logozzo and Manuel Fähndrich. 2008. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. In *ACM Symposium on Applied Computing*. 184–188.

Antoine Miné. 2002. *A Few Graph-Based Relational Numerical Abstract Domains.* 117–132.

Antoine Miné. 2006. The Octagon Abstract Domain. *Higher Order and Symbolic Computation* 19, 1 (2006), 31–100.

Antoine Miné, Enric Rodrguez-Carbonell, and Axel Simon. 2010. Speeding up Polyhedral Analysis by Identifying Common Constraints. *Electronic Notes in Theoretical Computer Science* 267, 1 (2010), 127 – 138.

Axel Simon and Andy King. 2005. Exploiting Sparsity in Polyhedral Analysis. In *Proc. Static Analysis Symposium (SAS)*. 336–351.

Axel Simon and Andy King. 2010. The Two Variable Per Inequality Abstract Domain. *Higher Order Symbolic Computation (HOSC)* 23 (2010), 87–143.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2015. Making Numerical Program Analysis Fast. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*. 303–313.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast Polyhedra Abstract Domain. In *Proc. Symposium on Principles of Programming Languages (POPL)*. 46–59.

Arnaud Venet and Guillaume Brat. 2004. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Proc. Programming Language Design and Implementation (PLDI)*. 231–242.

H. Le Verge. 1994. *A note on Chernikova's Algorithm.* Technical Report.